ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

COMPORTAMIENTO DEL PROTOCOLO QUIC SOBRE CANALES INALÁMBRICOS

(QUIC protocol behavior over wireless channels)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Álvaro Diego Bocos

Septiembre-2019

Índice general

Índice de figuras

Índice de tablas

Acrónimos

CHLO	Cliento Hello message.
FEC	Forward Error Correction.
GE	Gilbert-Elliott.
HTTP	Hyper Text Transfer Protocol.
HTTP2	Hyper Text Transfer Protocol version 2.
IP	Internet Protocol.
NACK	Negative Acknowledgement.
NS3	Network Simulator version 3.
QUIC	Quick UDP Internet Connections.
REJ	Rejection message.
RTT	Round-trip time.
SPDY	SPeeDY.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.

Resumen

En este documento se presenta un estudio sobre el comportamiento de QUIC y rQUIC sobre canales inalámbricos, configurados para emular dos modelos de errores: (1) uniforme y (2) a ráfagas.

Hoy en día, el crecimiento de la utilización de las redes inalámbricas ha hecho que dentro de la comunidad investigadora surja un gran interés por la búsqueda de nuevos protocolos que mejoren los anteriores. Uno de los aspectos que suscita mayor atención es la reducción de la latencia, siendo este una de las principales motivaciones por las que se propone QUIC.

Por ello, este estudio se centra en el análisis del tiempo requerido para la transmisión entre un cliente y un servidor QUIC, situados en dos contenedores LXC, sobre canales inalámbricos, haciendo uso del simulador de redes de tráfico NS3.

Abstract

This document presents a study on the performance of QUIC and rQUIC over wireless channels. These channels are configured to mimic two error models: (1) PER and (2) bursty errors.

Today, the growth in the use of wireless networks has led to a great interest within the scientific community to seek novel protocols that outperform existing ones. One of the aspects that gathers more interest is the reduction of latency, this being one of the main motivations behing QUIC development.

Therefore, this study focuses on the analysis of the time required to transmit a file between a client and a QUIC server, located in two LXC containers, over errors-prone wireless channels explained above, by means of network the NS3 traffic simulator.

Capítulo 1

Introducción

1.1. Motivación

Se desea reducir la latencia a través de Internet, proporcionando al usuario un conjunto de interacciones más "realistas". Con el tiempo, el ancho de banda en todo el mundo crecerá, pero los tiempos de ida y vuelta, limitados en última instancia por la velocidad de la luz, no disminuirán. Se necesita un protocolo que minimice latencia a través de Internet, que requiera menos retransmisiones.

Las dos motivaciones principales es reducir la latencia, soportando mejor SPDY, y juntar una mayor cantidad de tráfico.

SPDY es un protocolo de flujo multiplexado implementado actualmente sobre Transmission Control Protocol (TCP). Puede reducir la latencia enviando todas las solicitudes lo antes posible, esto es sin esperar a que se completen los GET anteriores, y puede reducir la utilización del ancho de banda al comprimir parte del tráfico redundante. Se han encontrado con varios problemas a la hora de usar de manera eficiente los recursos al mismo tiempo que se reduce la latencia.

- El retraso de un solo paquete produce el bloqueo en la "cabecera" de un stream Head-of-line blocking (HOL). El retraso de un solo paquete hace que se pause todo el conjunto de streams SPDY, debido a que

TCP únicamente proporciona un interfaz de flujo.

- Utilización desfavorable de conegestion avoidance por TCP, lo que conlleva una reducción del ancho de banda y una sobrecarga de la latencia. Cuando se pierde un solo paquete de una conexión SPDY sobre TCP, la ventana de congestión para toda la conexión se reduce en un 50 %.
- Retrasos en la reanudación de la sesión Transport Layer Security (TLS). Un protocolo de enlace TLS toma al menos un Round-trip time (RTT) adicional antes que los datos se puedan transmitir con éxito. Este retraso es debido a la implementación de TLS y no se debe a requisitos funcionales de seguridad.

Los principales objetivos de Quick UDP Internet Connections (QUIC) son los siguientes:

1. Implementación generalizada en Internet.
2. Reducir el bloqueo HOL debido a la pérdida de paquetes. Se busca que la pérdida de un paquete generalmente no perjudique a otros flujos multiplexados.
3. Baja latencia. Tanto la reducción de la latencia de inicio de conexión, como la de retransmisión después de la pérdida de paquetes utilizando códigos de corrección de errores de reenvío (Forward Error Correction (FEC)).
4. Soporte para evitar la congestión, comparable con TCP.
5. Garantías de privacidad comparables a TLS sin requerir transporte o descrifrado en orden.
6. Escalado de requisitos de recursos confiables y seguros, tanto del lado del servidor como del lado del cliente.
7. Consumo reducido de ancho de banda y mayor capacidad de respuesta del estado del canal a través de la señalización unificada en todos los streams multiplexados [1].

1.2. Estructura

Tras la introducción realizada en este capítulo, el resto del documento sigue la siguiente estructura. En el capítulo 2 se describen los fundamentos teóricos de QUIC y rQUIC y los modelos de errores a ráfagas, concretamente el modelo de Gilbert-Elliott. En el capítulo 3, se explican detalladamente los escenarios implementados y que se analizarán posteriormente. A continuación, en el capítulo 4, se recapitulan los resultados obtenidos. Por último, se exponen las conclusiones de los resultados obtenidos.

Capítulo 2

Antecedentes

2.1. QUIC

QUIC es un protocolo desarrollado por Google que reduce la latencia en comparación con TCP y resuelve una serie de problemas de las capas de transporte y de aplicación. Surgió debido a la urgente necesidad de innovación de la capa de transporte, principalmente por las dificultades para desplegar nuevos protocolos sobre TCP. En la Figura 2.1 se observa en que capas se encuentra frente al modelo tradicional.

QUIC es similar a TCP + TLS + Hyper Text Transfer Protocol version 2 (HTTP2) pero implementado sobre User Datagram Protocol (UDP). Permite una serie de innovaciones y ventajas sobre TCP + TLS + HTTP2:

1. Latencia en el establecimiento de la conexión.
2. Control de congestión mejorado.
3. Multiplexación sin bloqueo HOL.
4. Forward Error Correction (FEC).
5. Migración de conexión.

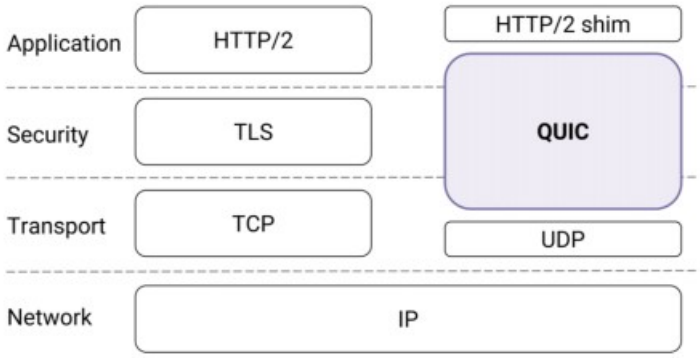


Figura 2.1: QUIC en comparación con la pila de protocolos tradicional

2.1.1. Establecimiento de la conexión

El principal motivo de un mejor rendimiento de QUIC en comparación con TCP es la velocidad de conexión. En primer lugar, de manera general, el intercambio para el establecimiento de la conexión en QUIC (Figura 2.3) necesita cero paquetes antes de enviar payload. Sin embargo, en TCP + TLS (Figura 2.2) son necesarios entre 1 y 3 paquetes.

Inicialmente, el cliente no sabe nada sobre el servidor. La primera vez que un cliente QUIC se conecta a un servidor, hace uso de un protocolo de un paquete para obtener toda la información necesaria para completar el protocolo de enlace (handshake). El cliente envía un saludo al servidor (Cliento Hello message (CHLO)) y este le responde con un rechazo (Rejection message (REJ)) con toda la información que el cliente necesita para continuar, incluyendo el token de la dirección origen y los certificados del servidor. La próxima vez que el cliente envíe un CHLO, puede usar las credenciales de la conexión anterior para enviar solicitudes encriptadas al servidor.

2.1.2. Formato de paquete

La unidad básica del protocolo QUIC es el paquete. Existen paquetes regulares que contienen frames, y paquetes especiales que pueden ser de negociación de versión o de reseteo.

La estructura de los paquetes en QUIC (Figura 2.5) comienzan con una

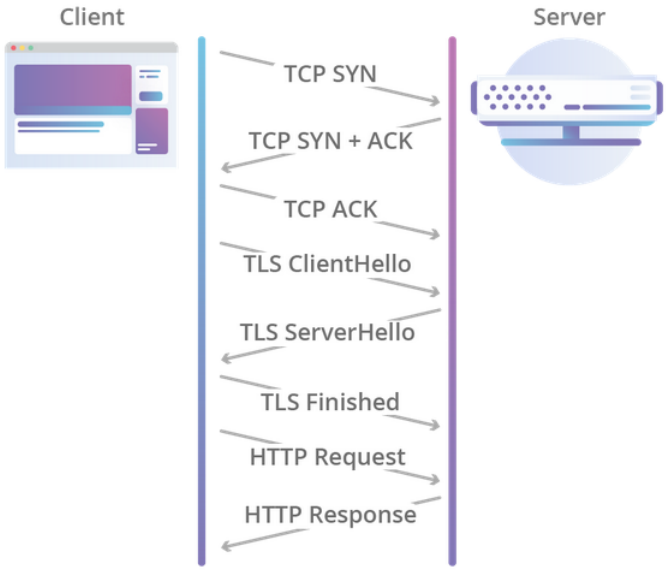


Figura 2.2: Petición de conexión HTTP sobre TCP + TLS

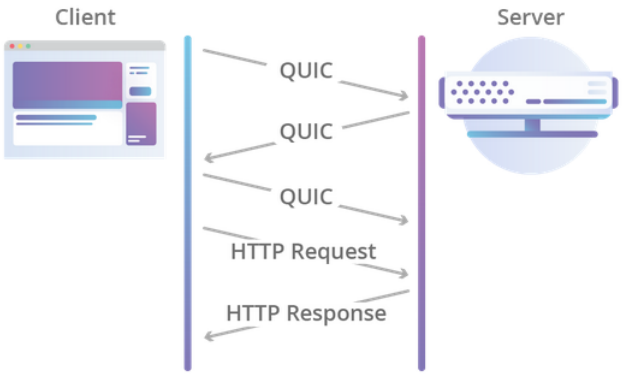


Figura 2.3: Petición de conexión HTTP sobre QUIC

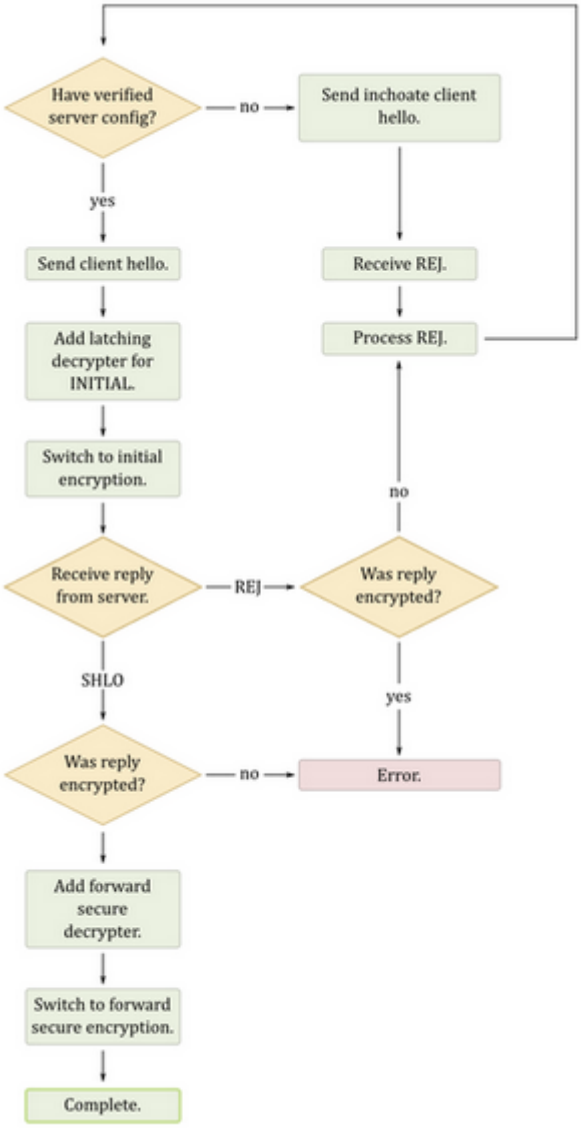


Figura 2.4: Protocolo de enlace del cliente

cabecera que ocupa entre 2 y 19 bytes, de los cuales uno es para flags públicas, otros 8 como máximo para el identificador de la conexión, 4 opcionales para la versión de QUIC, y por último, un máximo de 4 bytes para el número de paquete.

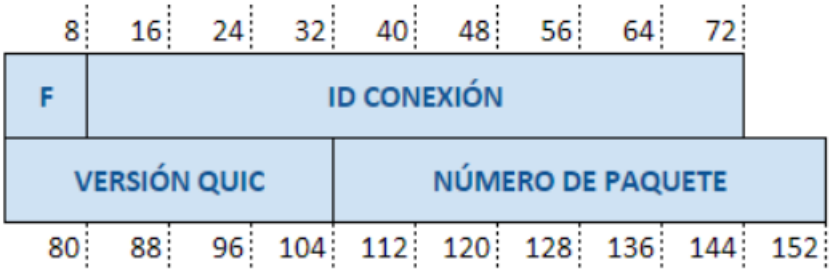


Figura 2.5: Formato de paquete QUIC

2.1.3. Control de congestión

QUIC ofrece un mecanismo de control de congestión con mayor información que el ofrecido por TCP originalmente.En ese sentido, QUIC está basado en una reimplementación de TCP Cubic.

En el modelo tradicional, utilizando HTTP2, dentro de una misma conexión TCP se permiten varias peticiones Hyper Text Transfer Protocol (HTTP). Esto da lugar a un problema denominado bloqueo HOL que consiste en que al hacerse varias peticiones sobre la misma conexión TCP todas se ven igualmente afectadas por la pérdida de paquetes. Para ello QUIC aporta como solución que se puedan asignar diferentes flujos HTTP a diferentes flujos de transporte QUIC. Esto da lugar a:

1. La conexión QUIC sigue siendo la misma, no hay protocolos adicionales y se comparte el estado de congestión.
2. Los streams QUIC se entregan de manera independiente, por lo que en la mayoría de los casos la pérdida de los paquetes en un stream no afecta a otros.
3. Reduce drásticamente el tiempo en redes altamente congestionadas con altas tasas de pérdidas.

Un ejemplo de la información adicional que aporta QUIC es que cada paquete lleva un número de secuencia nuevo, diferenciando entre el paquete original y el retransmitido. Esto permite que el emisor pueda distinguir los Acknowledgement (ACK) para transmisiones y retransmisiones, con lo cual

se termina el problema que sufre TCP con la ambigüedad en sus retransmisiones.

Además, QUIC soporta hasta 256 Negative Acknowledgement (NACK), lo que le hace más resistente al reordenamiento de bytes que TCP.

2.1.4. Multiplexación

Uno de los mayores problemas de HTTP2 es el bloqueo HOL en la parte superior de TCP. La aplicación ve la conexión TCP como un conjunto de bytes. Por tanto, cuando se pierde un paquete TCP, ningún stream en la conexión HTTP2 puede continuar hasta que el paquete se retransmite.

TCP utiliza los puertos TCP y las direcciones Internet Protocol (IP) de los sistemas conectados para identificar las conexiones. Esto hace que un cliente se pueda comunicar con el servidor a través de varios puertos utilizando la misma conexión. Sin embargo, QUIC recurre a una identificación de la conexión de 64 bits y a diferentes streams para transportar los datos de una conexión. Debido a que QUIC esta diseñado desde cero para la multiplexación, las conexiones QUIC no están vinculadas a un puerto específico, ya que se pueden modificar los puertos y las direcciones IP de la misma manera que las conexiones multiplexadas [2].

2.1.5. Corrección de errores

La corrección de errores de reenvío (FEC) permite que se transmitan bytes adicionales para proporcionar redundancia en caso de que no lleguen todos los paquetes. El FEC implementado por QUIC está basado en codificación XOR, utilizado para mejorar la confiabilidad de la capa de enlace.

El esquema FEC utiliza la operación lógica XOR para reconstruir un paquete cuando se pierde uno. El sistema tiene un concepto de grupo de paquetes, en el que cualquier paquete que se pierda se puede recuperar.

Debido a que los paquetes FEC forman parte de la carga de la red, QUIC los trata de la misma manera que los paquetes de datos desde el punto de vista de control de congestión. Esto significa que cuando se envía un paquete

FEC se consume ventana de congestión que podría usarse para transmitir paquetes de datos y si, en caso contrario, se pierde un paquete FEC, se produce una reducción de la ventana de congestión [3].

La principal ventaja de este tipo de FEC es que tiene una baja carga computacional y es sencillo de implementar. Sin embargo, solo permite recuperar un paquete. Por lo que si se pierden dos paquetes o más se termina desperdiciando el paquete FEC. A esto hay que añadirle la necesidad de indicar el número de grupo cuando se envía el paquete.

2.1.6. Migración de conexiones

El objetivo de QUIC es que en el momento que un cliente cambie la dirección IP, pueda seguir usando la identificación de la conexión anterior sin interrumpir ninguna solicitud en curso. Mientras que las conexiones TCP se identifican mediante el conjunto de dirección origen, puerto origen, dirección destino y puerto destino, las conexiones QUIC se identifican mediante una ID de conexión de 64 bits que se ha generado aleatoriamente por el cliente. Eso significa que si la identificación de la conexión para QUIC es independiente de cualquier dirección IP o cualquier puerto. Sin embargo, al utilizar TCP, si se cambia cualquier dirección IP o cualquier puerto, la conexión TCP activa ya no sería válida.

2.1.7. Control de flujo

QUIC proporciona un control de flujo similar al que proporciona HTTP2. Consiste en que el extremo de una conexión QUIC informa acerca de la cantidad de datos que esta dispuesto a recibir en cada stream.

El control de flujo en QUIC anuncia el desplazamiento de los bytes de la secuencia que el extremo está dispuesto a recibir. Cada vez que se libera memoria se envía un WINDOWUPDATE que le permite al transmisor enviar más datos y BLOCKED en el caso de que se bloquee por el control de flujo. Un ejemplo sería informar al extremo de que puede mandar hasta el byte 300 en cierta secuencia, y al haber enviado 200 bytes en esa secuencia, sabría que solo puede mandar 100 bytes más antes del bloqueo por el control de flujo.

Cada extremo elige de manera independiente el tamaño de la ventana de recepción de control de flujo máximo. Esta decisión se suele basar en los recursos disponibles.

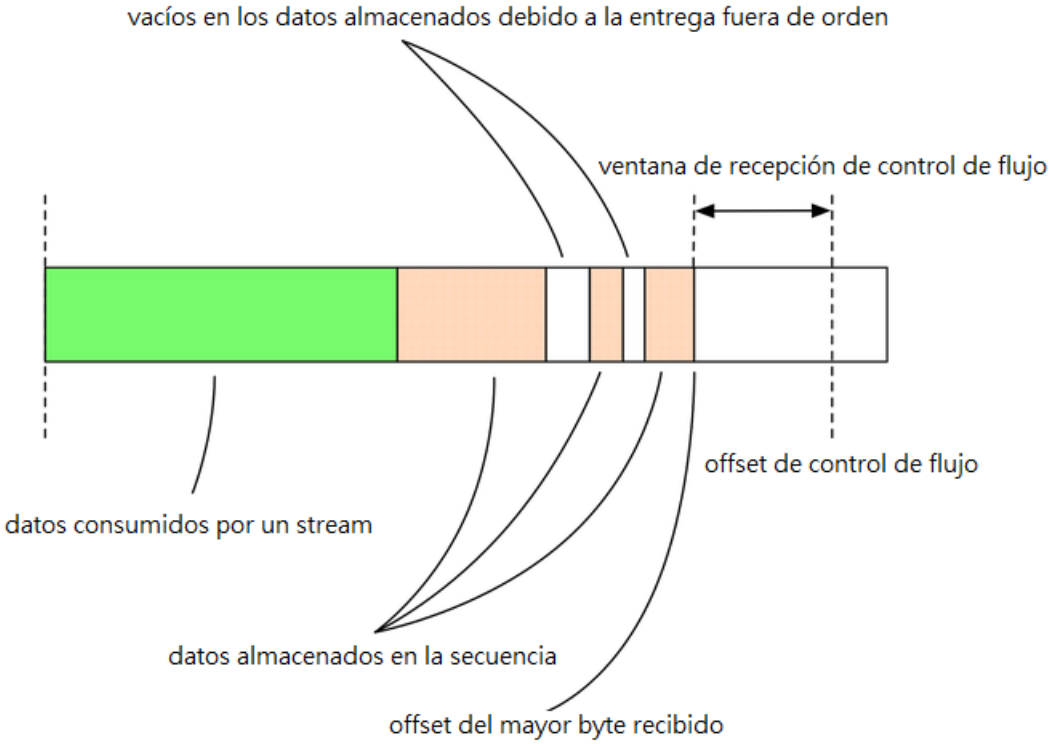


Figura 2.6: Ventana de recepción de control de flujo

En la figura 2.6, se muestra la situación en la que el peer ha enviado ciertos datos. Se puede observar que se han consumido algunos bytes y almacenado otros tantos. Además, hay huecos debido a que los paquetes han llegado fuera de orden. Es importante observar que la ventana de recepción se ha reducido en función del desplazamiento de los bytes recibidos y no de los bytes almacenados en el búfer. Según llegan nuevos frames se verifica si es necesario variar el offset, pudiendo únicamente aumentar y en el momento que una secuencia termina el peer debe informar sobre el offset del mayor byte en esa secuencia.

En el caso de que la diferencia entre el offset de la recepción del control de flujo y los bytes consumidos sea menor que la mitad de la ventana de recepción máxima se enviará un nuevo WINDOWUPDATE. Al enviarlo se elimina el offset de la recepción de control de flujo de manera que es igual a

la suma de los bytes consumidos y la ventana de recepción máxima.

Control de flujo de la conexión QUIC

El control de flujo en un stream no es suficiente para proteger al receptor de demasiados datos entrantes. El cliente abre el número máximo de streams posibles, que actualemente es 100, y envía el máximo de bytes que permite el control de flujo por stream en cada uno de los 100 streams. Por ese motivo, también se necesita que la conexión QUIC tenga control de flujo.

El control de flujo de una conexión QUIC funciona de la misma manera que en un stream, pero haciendo que los bytes consumidos y el offset más alto recibido sean agregados en todos los streams. De manera que cuando se actualiza el campo de bytes consumidos en un stream también se hace en el controlador de flujo a nivel de conexión [4].

2.2. RQUIC

rQUIC se diseñó con el objetivo de habilitar FEC dentro de QUIC.

En la Figura 2.7 se ven las modificaciones introducidas por rQUIC a partir de QUIC. Se generan paquetes QUIC que se cifran y autentican, y ese paquete cifrado se ingresa en el módulo rQUIC donde se añade el campo FEC a la cabecera QUIC y se aplica el algoritmo FEC correspondiente según el tipo de paquete [5].

2.2.1. Cabecera

La presencia de cabecera de FEC se puede indicar mediante flags en la cabecera QUIC o negociarse en el establecimiento de la conexión, debido a que en la sesión de QUIC se puede habilitar o deshabilitar rQUIC. El tamaño de la cabecera FEC es de 4 bytes y contiene los siguientes campos:

1. Tipo (1 byte): Indica si es paquete esta protegido por FEC (0x80), si

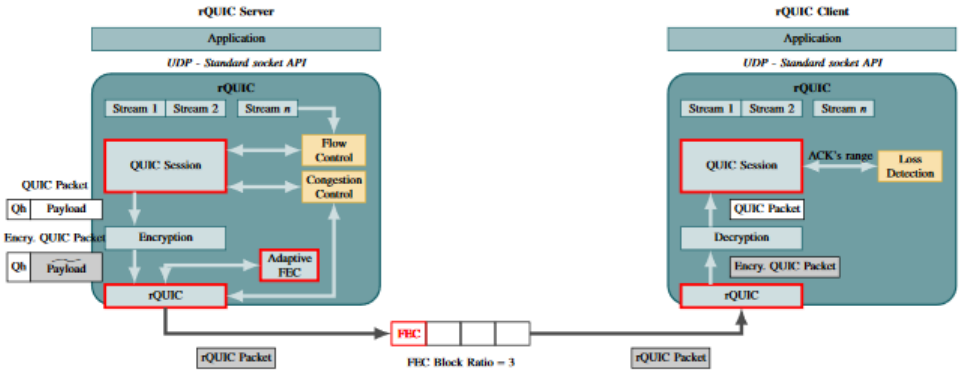


Figura 2.7: Componentes básicos de rQUIC

no lo esta (0x00), o si es un paquete FEC en si mismo (0xC0).

2. ID de bloque (1 byte): Identifica los paquetes protegidos por el mismo bloque.

3. FEC Ratio (1 byte): relación utilizada por el algoritmo FEC.

4. Count (1 byte): En el caso de que este protegido, este campo identifica el orden del paquete dentro del bloque FEC.

Los paquetes rQUIC son enviados respetando el control de flujo y congestión.

2.2.2. Algoritmo

Uno de los problemas que presentan los algoritmos de FEC a QUIC es ir en contra de la idea de mejorar la latencia para la codificación y la decodificación de QUIC. Por ello, se eligió un algoritmo FEC adaptativo basado en XOR que se explica a continuación.

1. FEC basado en XOR: Este FEC introduce un paquete adicional para cada n paquetes para proteger el bloque de datos, lo que quiere decir que introduce una redundancia de n+1. Ese paquete adicional es la XOR de los n paquetes de datos que hay que proteger. El enfoque

tan simple de usar una XOR presenta la gran ventaja de su baja complejidad computacional, pero también tiene la gran limitación de que solo se permitirá recuperar un paquete por bloque. Aún así, en caso de pérdidas múltiples, no impedirá el mecanismo clásico de recuperación de pérdidas con retransmisiones, aunque consumirá más recursos.

El tamaño de bloque FEC es un parámetro clave para la optimización, y por ese motivo se decide adoptar un algoritmo adaptativo que varía en el ratio FEC de acuerdo a los cambios en las retransmisiones que se necesitan.

2. FEC adaptativo: Las pérdidas en los canales inalámbricos pueden variar con el tiempo y eso da lugar a que sean difíciles de predecir. Por ese motivo, se implementa un FEC adaptativo que se basa en las pérdidas residuales, que son los paquetes que deben retransmitirse debido a que FEC no se recupera.

   La pérdida residual del total de paquetes transmitidos y retransmitidos durante un periodo, i, y con longitud, T, se calcula como:

$$\tau_i = \frac{retransmisiones}{transmisiones + retransmisiones} \tag{2.1}$$

   Las pérdidas residuales se promedian a lo largo de N períodos:

$$\overline{\tau_i} = \frac{\sum_{i=1}^{N} C_i}{N} \tag{2.2}$$

   La manera de actualizar el FEC es la siguiente (ver Algoritmo 1): si el promedio residual es mayor de un valor γ , y el FEC aumenta en δ y disminuye de la misma manera en caso contrario.

Algoritmo 1 Algoritmo de rQUIC con FEC adaptativo

Parametros: $r = r_{init}$

1: **if** $\overline{\tau} > \gamma$ **then**
2: **return** $r = r \cdot (1 - \delta)$
3: **else**
4: **return** $r = r \cdot (1 + \delta)$
5: **end if**

2.3. Errores a ráfagas

Un error a ráfaga es una secuencia continua de símbolos, recibida a través de un canal de comunicación, de forma que el primer y el último símbolo tienen un error y existe una secuencia de m símbolos recibidos correctamente dentro de esa ráfaga de error. Ese parámetro m se denomina banda de guarda [6].

2.3.1. Modelo de Gilbert-Elliott

El modelo de canal de utilizado que se ha utilizado es el modelo de Gilbert-Elliot. Este es un modelo para describir patrones de error de ráfaga en canales de transmisión. Se basa en una cadena de Markov (Figura 2.8) donde hay dos estados: Good, donde se ha transmitido correctamente y Bad, en el caso contrario. Se interpreta un evento como la llegada de un paquete, y un error como la pérdida de paquete [7].

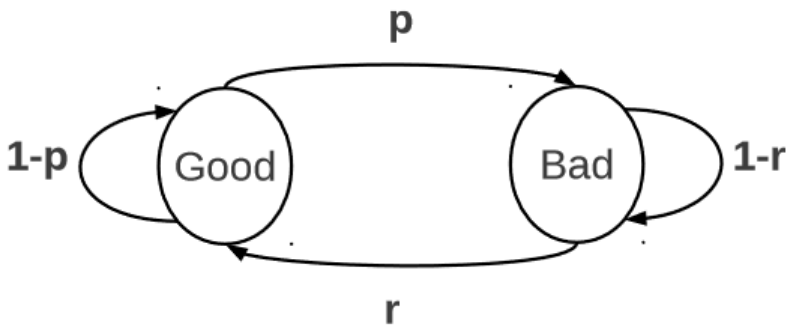


Figura 2.8: Cadena de Markov de Gilbert-Elliot

Una transición del estado Good al estado Bad tiene lugar cada vez que el paquete actual se pierde cuando el anterior llego de manera exitosa. En el caso opuesto, se produce una transición del estado Bad al estado Good cada vez que se recibe con éxito el paquete actual, pero el anterior se ha perdido. La probabilidad de pérdida condicionada a que el estado anterior sea exitoso es denotada como p mientras que la probabilidad de pérdida condicionada a que el estado anterior fue Bad es denotada como 1-r. Finalmente el modelo de GE queda definido por la siguiente matriz de transición [8]:

$$M = \begin{pmatrix} 1-p & p \\ r & 1-r \end{pmatrix} \tag{2.3}$$

Partiendo del Teorema de Probabilidad Total se obtiene la probabilidad de estar en cada uno de los dos estados:

$$Prob\{Good\} = Prob\{Good\} \cdot Prob\{Good|Good\} + Prob\{Bad\} \cdot Prob\{Good|Bad\} \rightarrow \tag{2.4}$$

$$\rightarrow Prob\{Good\} = Prob\{Good\} \cdot (1-p) + Prob\{Bad\} \cdot r \tag{2.5}$$

$$Prob\{Bad\} = Prob\{Bad\} \cdot Prob\{Bad|Bad\} + Prob\{Good\} \cdot Prob\{Bad|Good\} \rightarrow \tag{2.6}$$

$$\rightarrow Prob\{Bad\} = Prob\{Bad\} \cdot (1-r) + Prob\{Good\} \cdot p \tag{2.7}$$

Resolviendo un sistema con las dos ecuaciones resultantes anteriores se llega a las dos probabilidad de los estados:

$$Prob\{Good\} = \frac{r}{r+p} \tag{2.8}$$

$$Prob\{Bad\} = \frac{p}{r+p} \tag{2.9}$$

Teniendo en cuenta las probabilidades de los estados, la probabilidad de pérdida se define como:

$$Prob\{Perdida\} = Prob\{Bad\} \cdot (1-r) + Prob\{Good\} \cdot p = Prob\{Bad\} \tag{2.10}$$

Capítulo 3

Implementación

3.1. Creación de la topología

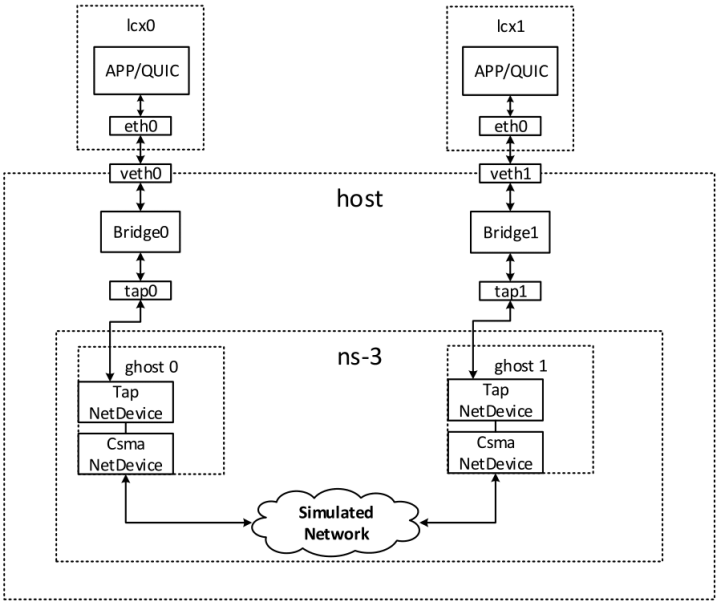


Figura 3.1: Topología

Inicialmente, se exponen de manera breve los principales comandos básicos que se usan para llevar a cabo el despliegue de la plataforma experimnetal en Network Simulator version 3 (NS3):

- Mostrar bridges/interfaces.

```
sudo brctl show
```

- Iniciar contenedores LXC

```
sudo lxc-start -n contenedor -d
```

- Parar contenedores LXC

```
sudo lxc-stop nombre
```

- Información de los contenedores

```
sudo lxc-ls -f
```

- Eliminar bridge

```
sudo brctl delbr nombre
```

- Eliminar tap

```
sudo ip link delete tap-nombre
```

- Levantar/tirar un interfaz

```
sudo ifconfig nombre up/down
```

- Eliminar contenedor

```
sudo lxc-destroy -n nombre
```

En primer lugar, es necesario crear dos contenedores, que se han denominado “left” y “right”, en los que estará instalado QUIC y rQUIC. Para ello es necesario un fichero de configuración para cada uno de los dos contenedores, con el contenido que se muestra en las Figuras 3.2 y 3.3.

Como se puede observar, los campos definidos en los ficheros de configuración son:

1. “lxc.uts.name” define el nombre del contenedor. Además, el nombre del dispositivo tap en el host tendrá asignado como nombre “tap-nombredelcontedor” en este caso “tap-left” y “tap-right”.

```
#Container with network virtualized using a pre-configured bridge named br-source and
# veth pair virtual network devices
lxc.uts.name = left
lxc.net.0.type = veth
lxc.net.0.flags = up
lxc.net.0.link = br-left
lxc.net.0.ipv4.address = 10.0.0.1/24
```

Figura 3.2: Fichero de configuración lxc-left.conf

```
#Container with network virtualized using a pre-configured bridge named br-source and
# veth pair virtual network devices
lxc.uts.name = right
lxc.net.0.type = veth
lxc.net.0.flags = up
lxc.net.0.link = br-right
lxc.net.0.ipv4.address = 10.0.0.2/24
```

Figura 3.3: Fichero de configuración lxc-right.conf

2. "lxc.net.i.type" define el tipo de red usada por el contenedor, en este caso ambas usan virtual ethernet (veth), al ser dos redes distintas, el indice i es 0 y 1, respectivamente

3. "lxc.net.i.link" define el nombre del bridge al que se conecta el contenedor. Además, ese bridge tiene añadidos los taps para interaccionar con NS3.

4. "lxc.net.i.ipv4.address" para especificar la dirección IPv4 del contenedor. Será la misma que la del nodo ghost del NS3 vinculado al contenedor correspondiente que adquiere al usar la clase TapBridge con el modo UseBridge.

A continución, se crearon los bridges que utilizarán los paquetes para entrar y salir de los contenedores y los tap que utilizará NS3 para obtener los paquetes de los bridges a los que se hace referencia en los archivos de configuración. Para ello sera necesario ejecutar los comandos de la Figura 3.4.

```
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo brctl addbr br-left
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo brctl addbr br-right
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo tunctl -t tap-left
Set 'tap-left' persistent and owned by uid 0
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo tunctl -t tap-right
Set 'tap-right' persistent and owned by uid 0
```

Figura 3.4: Creación de bridges y tap

Antes de agregar los dispositivos tap a los bridges se deben configurar sus direcciones IP en 0.0.0.0, y abrirlos con los comandos de la Figura 3.5.

```
administrator@alvaro-diego:~$ sudo ifconfig tap-left 0.0.0.0 promisc up
administrator@alvaro-diego:~$ sudo ifconfig tap-right 0.0.0.0 promisc up
```

Figura 3.5: Configuración de la direcciones IP

Lo siguiente es agregar los tap a los respectivos bridges y abrirlos mediante los comandos que se muestran en la Figura 3.6.

```
administrator@alvaro-diego:~$ sudo brctl addif br-left tap-left
administrator@alvaro-diego:~$ sudo ifconfig br-left up
administrator@alvaro-diego:~$ sudo brctl addif br-right tap-right
administrator@alvaro-diego:~$ sudo ifconfig br-right up
```

Figura 3.6: Agregación de los tap a sus respectivos bridges

A continuación, se puede comprobar que los dos bridges están relaciones con sus respectivos tap como interfaces. Esto se puede comprobar en la Figura 3.7.

```
administrator@alvaro-diego:~$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
br-left         8000.1a97215c3bf9       no              tap-left
br-right                8000.aa46670c1956       no              tap-right
lxcbr0          8000.00163e000000       no
```

Figura 3.7: Comprobación de la agregación de los bridges a su respectivo tap

Por último, se crean ambos contenedor a partir de sus respectivos ficheros de configuración, y se inician todo a partir de los comandos de la Figura 3.8, tal y como se comprueba en la Figura 3.9.

```
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo lxc-create -f lxc-left.conf -t download -n left -- -d ubuntu -r trusty -a amd64
Using image from local cache
Unpacking the rootfs

---
You just created an Ubuntu trusty amd64 (20190814_07:42) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo chroot /var/lib/lxc/left/rootfs/ passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo lxc-start -n left
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo lxc-create -f lxc-right.conf -t download -n right -- -d ubuntu -r trusty -a amd64
Using image from local cache
Unpacking the rootfs

---
You just created an Ubuntu trusty amd64 (20190814_07:42) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo chroot /var/lib/lxc/right/rootfs/ passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo lxc-start -n right
```

Figura 3.8: Creación de los contenedores

```
administrator@alvaro-diego:~/workspace/ns-allinone-3.29/ns-3.29/src/tap-bridge/examples$ sudo lxc-ls -f
NAME  STATE   AUTOSTART GROUPS IPV4     IPV6 UNPRIVILEGED
left  RUNNING 0         -      10.0.0.1 -    false
right RUNNING 0         -      10.0.0.2 -    false
```

Figura 3.9: Comprobación de la creación correcta de los contenedores

Una vez se han creado los contenedores según la topología de la Figura 3.1 es necesario instalar QUIC dentro de los mismos. Para ello, se usan carpetas compartidas para que dentro de los contenedores está todo lo necesario para su instalación.

Para ello es necesario añadir las siguientes líneas (Figuras 3.10 y 3.11) en cada uno de los ficheros de configuración.

```
lxc.mount.entry=/home/administrator/go /var/lib/lxc/left/rootfs/go none bind 0 0
```

Figura 3.10: Carpeta compartida contenedor left

```
lxc.mount.entry=/home/administrator/go2/go /var/lib/lxc/right/rootfs/go none bind 0 0
```

Figura 3.11: Carpeta compartida contenedor right

También es necesario añadir esas líneas dentro del fichero “config” situado en los directorios “/var/lib/lxc/left” y “/var/lib/lxc/right”. Además también es necesario que dentro de los contenedores se creen las carpetas “go” y “go2”, que son los nombres que se han asignado a las carpetas compartidas y que se reinicien los contenedores.

Una vez que ya están creados los contenedores, se necesitar instalar go en ellos. Para ello se ejecutan los comandos de la Figura 3.12.

```
root@left:~/go# sudo tar -C /usr/local -xzf go1.9.2.linux-amd64.tar.gz
root@left:~/go# echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.profile
root@left:~/go# source ~/.profile
root@left:~/go# go version
go version go1.9.2 linux/amd64
```

Figura 3.12: Instalación de Go dentro de los contenedores

3.2. Implementación del escenario

En este capítulo se explica la implementación del enlace entre los dos contenedores según el modelo de errores de Gilbert-Elliott (GE) explicado anteriormente en el capítulo 2. Esta implementación se lleva a cabo a través del programa NS3, con ficheros de C++.

3.2.1. Simulador NS3

NS3 es un simulador de redes basado en eventos discretos y en una librería C++ que proporciona diferentes modelos de simulación de redes implementados a partir de objetos C++. Para interaccionar con la esta librería se hace mediante aplicaciones C++ que instancian unos modelos de simulación para desplegar el escenario requerido, entrar en el bucle principal y salir cuando la simulación se completa.

Namespace NS3

La función de los namespaces es definir los diferentes contextos en los que se desarrolla el código. Los proyectos de este software se implementan en un namespace llamado NS3. Esto lo que hace es agrupar todas las declaraciones que tienen relación con NS3 dentro del contexto global en el que se encontraría en el caso de no especificar ningún NS3.

El uso de este espacio ahorra la necesidad de introducir 'ns3::' cada vez que se haga alusión a un elemento definido por el software.

Módulos

La estructura de este software se caracteriza por ser modular, es decir, que está estructurada en módulos. Dentro de cada uno de estos módulos, están definidas las diferentes clases. El interés de repartirlo en módulos es que la gran mayoría de las clases tienen dependencia de otras, por lo que sería necesario incluir todos los archivos en la cabecera. Todos los módulos que existen dentro de NS3 se recogen en la Tabla 3.1.

Tabla 3.1: Listado de módulos de NS3

6LoWAPN	AODV Routing
Applications	BRITE Topology Generator
Bridge Network Device	Carrier-Sense Multiple Access (CSMA) Layaout Helpers
CSMA Network Device	Click Routing
Configuration Store/load Constants	Core
Destination-Sequenced Distance-Vector (DSDV) Routing	Data Set Ready (DSR) Routing
Energy Models	File Descriptor Network Device
Flow Monitor	Internet
Internet Applications	LR-WAPN models
Long Term Evolution (LTE) Models	MPI Distributed Simulation
Mesh Device	Mobility
Network	Network Animation
Nix-Vector Routing	OLSR Routing
OpenFlow Switch Device	Point-To-Point Network Device
Point-to-Point Layout Helpers	Propagation Models
Spectrum Models	Statistics
Tap Bridge Network Device	Topology Input Readers
Traffic-control	UAN Models
Utils	Virtual Device
Visualizer	WAVE Module
WiMAX Models	Wifi Models

3.2.2. Modelo de errores

Como se explica en el Capítulo 2 el modelo de errores utilizado es el modelo de Gilbert-Elliott. Para su implementación dentro de NS3 se crea una clase Gilbert-Elliott, heredada de una clase general de NS3 para los modelos de errores llamada ErrorModel. El pseudocódigo utilizado para su implementación es el que está representado en el Algoritmo 2.

Antes de implementar en Algoritmo 2, a través de la función GetSerializedSize de NS3 que devuelve el número de bytes necesarios para la serialización de paquetes, se filtran los paquetes de QUIC o de rQUIC, que necesitan 128 ó 380 y 1308 bytes respectivamente.

Tanto value1 como value2, que aparecen en el Algoritmo 2, son dos números generados a partir de una variable aleatoria uniforme entre los valores 0 y 1. Esto se puede hacer a partir de una clase llamada UniformRandomVariable, que permite crear un objeto para sacar valores entre unos valores mínimo y máximo que se consideren necesarios.

Las últimas variables de entrada son las dos probabilidades de transición entre los estados que se introducen desde el programa principal. La variable

Algoritmo 2 Modelo de Gilbert-Elliot

Parametros: El estado del paquete anterior, dos números aleatorios generados a partir de una variable aleatoria uniforme y las probabilidades de transición entre estados 'r' y 'q'

Salida: El estado del paquete que acaba de llegar

```
1: if anterior == false then
2:    if value2 < (1 - q) then
3:       return  anterior = true
4:    end if
5: else
6:    if value1 ⩽ r then
7:       return  anterior = false
8:    end if
9: end if
```

r es la correspondiente al paso del estado Good a Bad, y la variable q es la correspondiente al caso contrario.

La tasa de error de paquete (PER) se usa para comprobar el rendimiento del receptor y se calcula como el cociente de el número de paquetes incorrectamente recibidos entre el total de todos los enviados. Esto se puede ver en la fórmula 3.1.

$$PER = \frac{paquetes\ incorrectos}{paquetes\ correctos + paquetes\ incorrectos} \tag{3.1}$$

Partiendo de la expresión del número medio de paquetes, que se corresponde con una varible aleatoria geométrica, dada en la ecuación 3.2, se puede llegar a las expresiones del número medio de paquetes en cada uno de los dos estados de la cadena de markov del modelo.

$$\overline{N} = \sum_{i=0}^{\infty} i \cdot P_i \tag{3.2}$$

Para poder utilizar la expresión 3.2 se necesita saber la probabilidad de estar en cada uno de los estados. Asumiendo que siempre se comienza en el estado Good, la probabilidad de que un número de paquetes, i, que se reciben consecutivamente de manera correcta viene dada por la ecuación 3.3

y análogamente los i paquetes que se reciben incorrectamente vienen definidos por la expresión 3.4.

$$\#paquetes\ good\ consecutivos\ =\ (1-r)^{i-1}\cdot r \tag{3.3}$$

$$\#paquetes\ bad\ consecutivos\ =\ (1-q)^{i-1}\cdot q \tag{3.4}$$

Teniendo en cuenta lo explicado anteriormente, llegamos a las dos expresiones del número de paquetes medio del estado Good en función de las probabilidades de transición (ecuación 3.5).

$$\begin{aligned}
&\overline{N_g} = \sum_{i=0}^{\infty} i\cdot(1-r)^{i-1}\cdot r \rightarrow \overline{N_g} = r\cdot\sum_{i=0}^{\infty} i\cdot(1-r)^{i-1} \rightarrow \\
&\rightarrow \overline{N_g} = r\cdot\frac{1}{(1-(1-r))^2} \rightarrow \overline{N_g} = \frac{r}{r^2} \rightarrow \\
&\rightarrow \overline{N_g} = \frac{1}{r}
\end{aligned} \tag{3.5}$$

Y análogamente, se obtiene el número medio de paquetes en el estado Bad en función de las probabilidades de transición (ecuación 3.6).

$$\begin{aligned}
&\overline{N_b} = \sum_{i=0}^{\infty} i\cdot(1-q)^{i-1}\cdot q \rightarrow \overline{N_b} = q\cdot\sum_{i=0}^{\infty} i\cdot(1-q)^{i-1} \rightarrow \\
&\rightarrow \overline{N_b} = q\cdot\frac{1}{(1-(1-q))^2} \rightarrow \overline{N_b} = \frac{q}{q^2} \rightarrow \\
&\rightarrow \overline{N_b} = \frac{1}{q}
\end{aligned} \tag{3.6}$$

Para calcular la PER experimentalmente en nuestro escenario, simplemente se utiliza un contador que recoja la cantidad de paquetes que llegan a cada uno de los estados y se aplica la ecuación 3.1. Teniendo en cuenta las probabilidades de transición que se han explicado anteriormente y sabiendo que el número medio de paquetes en cada estado se corresponde con las ecuaciones 3.5 y 3.6 se puede obtener, a partir de ellas y de la ecuación 3.1, la siguiente expresión 3.7.

$$PER = \frac{\frac{1}{q}}{\frac{1}{q} + \frac{1}{r}} \rightarrow PER = \frac{r}{r+q} \tag{3.7}$$

3.2.3. Enlace

La conexión entre NS3 y los contenedores es posible gracias al modelo TapBridge implementado en el simulador, que conecta las entradas y salidas de un dispositivo de red del NS3 con las entradas y salidas del tap que es el dispositivo de red de Linux que se usa para crear bridges.

El resultado de esta conexión es la integración de dos contenedores Linux Container (LXC), que soporta dispositivos tap, con las simulaciones sobre NS3.

En el siguiente fragmento de código se puede ver como se ejecuta esto mediante C++ en NS3:

```
NodeContainer nodes;
nodes.Create(2);
NS_LOG_INFO("Nodes created");

TapBridgeHelper tapBridge;
tapBridge.SetAttribute("Mode", StringValue("UseBridge"));
tapBridge.SetAttribute("DeviceName", StringValue("tap-left"));
tapBridge.Install(nodes.Get(0), devices.Get(0));

tapBridge.SetAttribute("DeviceName", StringValue("tap-right"));
tapBridge.Install(nodes.Get(1), devices.Get(1));
```

Para trabajar con sistemas reales dentro de NS3 y poder transmitir y recibir paquetes se utiliza un planificador de tiempo real para sincronizar la simulación con el reloj de la máquina llamado RealTimeScheduler. La función del planificador es conseguir que el reloj de la simulación vaya a la misma velocidad que el del ordenador.

La implementación del RealTimeScheduler en NS3 es la siguiente:

```
GlobalValue::Bind("SimulatorImplementationType", StringValue("
    ns3::RealtimeSimulatorImpl"));
GlobalValue::Bind("ChecksumEnabled", BooleanValue(true));
```

Si no se ejecuta el RealTimeScheduler, el simulador aumenta el tiempo de simulación hasta el siguiente evento programado y durante la ejecución del evento el tiempo de simulación se para. La diferencia con el RealTimeScheduler es que, aunque el comportamiento es similar, el simulador intenta mantener el reloj de la simulación sincronizado con el del ordenador.

Al terminar la ejecución de un evento, el RealTimeScheduler chequea el tiempo de simulación del siguiente evento y lo compara con el del ordenador y, si es mayor, el simulador permanece en espera hasta que alcanza el tiempo real.

La variable que principalmente se quiere estudiar es el Download Complete Time (DCT), el tiempo total de descarga de un fichero completo. Para poder monitorizar variables NS3 ofrece la utilización de trazas.

En el caso particular de este trabajo, la variable está definida en "GilbertElliot.h", donde se encuentra el modelo de errores y ese mismo script se va calculando el valor de la variable que nos interesa. Mediante el uso de trazas se puede obtener el valor del tiempo cada vez que cambia de estado, y guardarlo en un fichero de datos. También se monitoriza de manera análoga el Packet Error Rate (PER).

La definición de la variable dentro del script que contiene el modelo de errores es la siguiente:

```
TypeId Gilbert_Elliot::GetTypeId(void) {
        static TypeId tid = TypeId("ns3::GilbertElliot")
                                    .AddTraceSource("DCT", "Download
                                        _Complete_Time",
                MakeTraceSourceAccessor(&Gilbert_Elliot::m_dct),
                "ns3::TracedValueCallback::Double")
                ;
        return tid;
}
```

A continuación, se generan las trazas en el programa principal y se define un stream vinculado a un fichero con extensión .txt para guardar los datos.

```
Ptr<OutputStreamWrapper> stream4 = asciiTraceHelper.
    CreateFileStream ("GEdct.txt");
pem->TraceConnectWithoutContext ("DCT", MakeBoundCallback (&
    DCTTrace, stream4));
```

Por último, se crea una función que simplemente escriba los datos dentro del fichero que se ha seleccionado anteriormente, cuya entrada siempre serán por defecto las mismas: el stream vinculado al archivo donde se va a guardar, el valor anterior y el nuevo valor que ha adquirido la variable.

```
void
DCTTrace(Ptr<OutputStreamWrapper> stream4, double oldValue,
    double newValue) {
     //*stream4->GetStream () << "DCT: " << newValue << " ms";
     *stream4->GetStream () << "DCT " << newValue << " ms" << std
         ::endl;
}
```

3.2.4. Contenedores LXC

Como ya se ha explicado anteriormente, la manera de que dentro de los contenedores esté todo lo relacionado con QUIC y con Go es a través de carpetas compartidas. Todo ese contenido se coloca dentro de los directorios /home/administrator de cada uno de los contenedores. Una vez ahí se procede a instalar Go, aunque surge el inconveniente que, cada vez que se sale del contenedor y se vuelve a entrar, la instalación ha desaparecido. Por tanto, es necesario repetir el proceso cada vez que se entre a los contenedores. Es por ello que se creó un archivo "install.sh"(Figura 3.13) que facilita y automatiza la instalación de Go.

```
#!/bin/bash
echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.profile
source ~/.profile
go version
```

Figura 3.13: Contenido de install.sh

Ficheros QUIC

Para las pruebas realizadas con QUIC hay dos ficheros principales: sink_server.go y bulk_client.go. El primero de ellos es un servidor que recibe todo lo que manda el segundo, que es el cliente, que transmite todos los paquetes con información aleatoria que puede de tamaño 1400 bytes.

Además, cada uno de los dos calcula el throughput en Mbps que se define como la cantidad de información transmitida de forma efectiva en un enlace de comunicaciones o como la velocidad de transferencia neta de la red que se calcula mediante la expresión 3.8.

$$Throughput\,[Mbps] = \frac{Informacion\ util\,[bytes]}{Tiempo\,[ms]} \tag{3.8}$$

Inicialmente se ejecuta el servidor que escucha en la dirección IP, que es la misma que la del propio contenedor donde se esta ejecutando, y puertos especificados una conexión QUIC. Posteriormente, genera una configuración TLS y vuelve a escuchar en los mismos: dirección IP y puerto, una conexión TCP + TLS y, una vez establecida la conexión, el servidor espera a recibir paquetes. Por último calcula el Throughtput según el Algoritmo 3.

Algoritmo 3 sink_server.go(Throughput)

Parametros: La dirección IP del contenedor donde se está ejecutando el servidor

Salida: El estado del paquete que acaba de llegar

```
1: for  do
2:    n = bytes(Stream)
3:    if error == false then
4:       t = get.Time()
         Thput = bytesReceived · 8 / t
5:    end if
6: end forbytesReceived = bytesReceived + n
```

Una vez que el servidor está ejecutado, se lanza el cliente, que recibe como argumento de entrada la misma dirección IP y el mismo puerto que la del contenedor donde se está ejecutando el servidor. Genera mensajes aleatorios con un tamaño de paquete especificado que en este caso, son 1400 bytes. Además, también calcula el throughtput, de una manera similar al servidor.

Capítulo 4

Análisis de resultados

En este capítulo se explica el proceso de recogida de las medidas necesarias para analizar el rendimiento de los algoritmos expuestos en los capítulos anteriores y comentar los mismos.

4.1. Proceso de medida

El escenario en el que se realizarán las medidas es el descrito en el capítulo anterior. Consta de dos contenedores unidos a través de un enlace CSMA, y las medidas se realizan a través de las herramientas proporcionadas por NS3.

A continuación, se llevarán a cabo la comparación de los resultados usando QUIC y rQUIC, diferenciando en este último caso entre FEC dinámico y estático. Además, se tendrá en cuenta dos modelos de errores: uno a partir de una PER uniforme y otro a ráfagas, según el modelo de Gilbert-Elliott. Los valores de PER utilizados serán de 0.01, 0.02, 0.03 y 0.05 y, para cada uno de ellos en los diferentes escenarios, se realizarán un total de 100 simulaciones.

Para el cálculo de las probabilidades de transición necesarias para conseguir las PER especificadas anteriormente se utilizó un script de Matlab fijando la PER y variando el número medio de paquetes en el estado Bad. El resultado de los valores que se van a utilizar está recogido en la Tabla 4.1.

Tabla 4.1: Valores de las simulaciones para Gilbert-Elliott

			N_b		
			5	10	20
PER	0.01	P_{gb}	0.002	0.001	0.000505
		P_{bg}	0.2	0.1	0.05
		N_g	495	990	1980
	0.02	P_{gb}	0.0041	0.002	0.001
		P_{bg}	0.2	0.1	0.05
		N_g	245	490	980
	0.03	P_{gb}	0.0062	0.0031	0.0015
		P_{bg}	0.2	0.1	0.05
		N_g	161.67	223.33	646.67
	0.05	P_{gb}	0.0105	0.0053	0.0026
		P_{bg}	0.2	0.1	0.05
		N_g	95	190	380

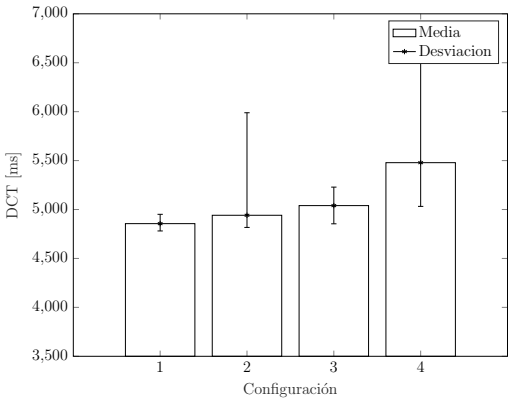
4.2. Resultados

En esta sección se analizan los resultados obtenidos de las simulaciones en los tres casos explicados anteriormente. El análisis está centrado en estudiar en DCT en cada uno de los casos.

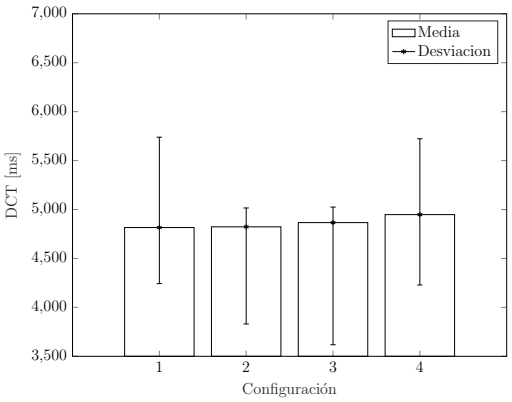
4.2.1. QUIC

En este apartado se analizan los resultados de QUIC con un modelo de PER uniforme y a ráfagas.

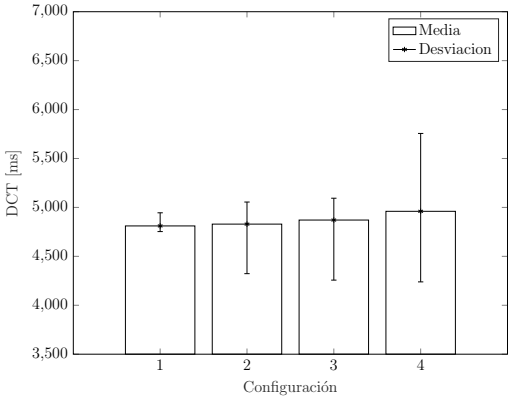
En las Figuras 4.1a, 4.1b, 4.1c y 4.1d se muestran el valor medio de las simulaciones, así como los valores máximos y mínimos obtenidos para los distintos valores de PER (1 %, 2 %, 3 % y 5 %) y las configuraciones empleadas. Cada una de las configuraciones se corresponden con los valores de PER en orden ascendente.



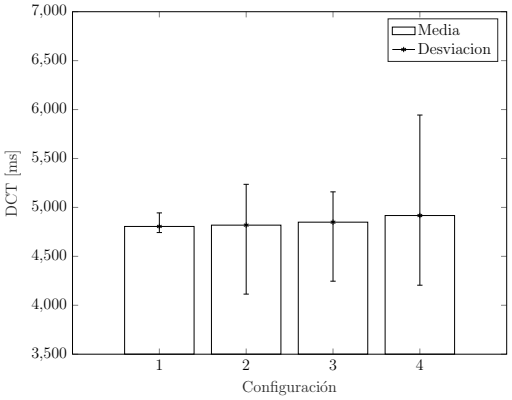
(a) QUIC con PER uniforme



(b) QUIC con PER a ráfagas $N_b = 5$



(c) QUIC con PER a ráfagas $N_b = 10$



(d) QUIC con PER a ráfagas $N_b = 20$

Comparando los valores obtenidos con el modelo a ráfagas para los distintos valores de PER y de número medio de paquetes recogidos en la Figura 4.2 se observa que cuanto más aumenta el número de paquetes mejores son los valores de DCT. Esta diferencia se hace más notoria en las tasas de PER más altas. En la Figura 4.2, las cofiguraciones referidas en el eje horizontal son para los valores 5, 10 y 20 de N_b.

Numericamente, se aprecia la diferencia en los valores con PER del 5 %. Para este caso en el caso de N_b=5 el DCT medio es de 4947.93 ms, mientras que en el caso de N_b=20 dicho valor es de 4916.98 ms lo que hace una diferencia de 30.95 ms, menos en el caso con mayor número de paquetes medios.

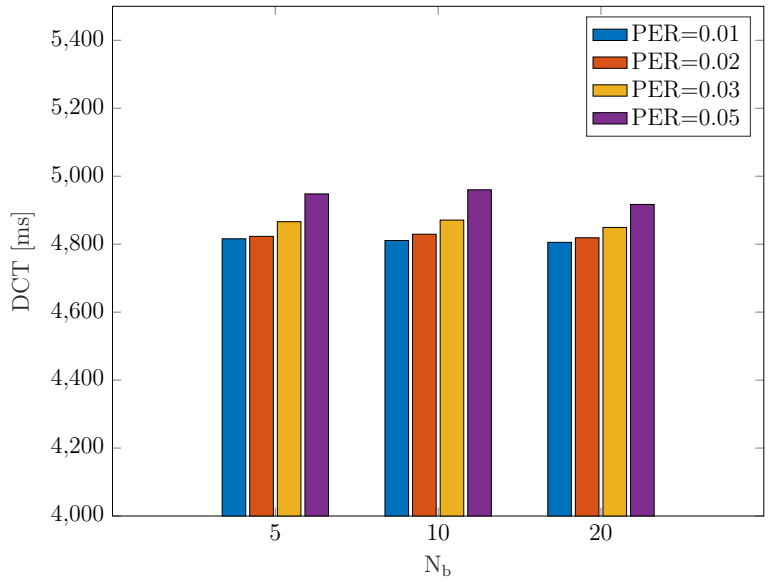


Figura 4.2: Comparación QUIC modelo a ráfagas para distintos PER

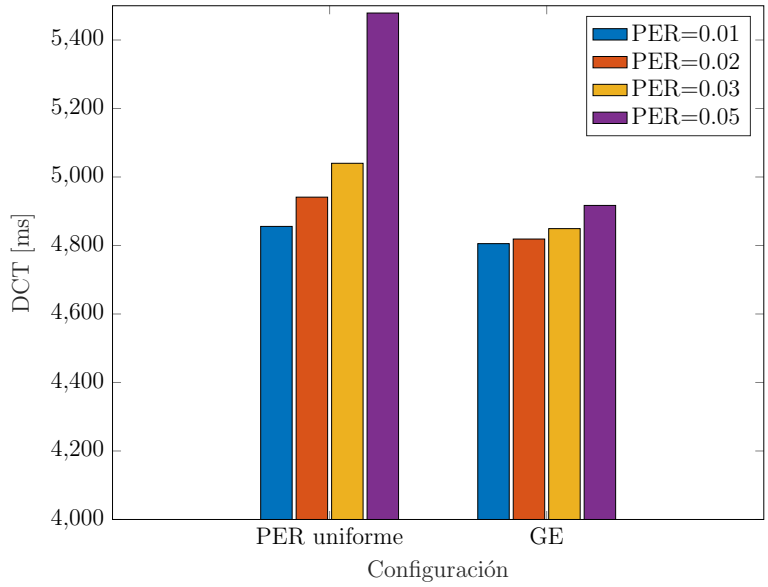


Figura 4.3: Comparación entre mejores valores de Gilbert-Elliot y PER uniforme

Comparando los valores obtenidos en un entorno con PER uniforme con los valores óptimos en el modelo a ráfagas se observa que disminuye de manera considerable sobre todo en los valores con PER del 5 %. En este caso se reduce

desde 5478.74 ms, con PER uniforme hasta 4916.98 siendo la diferencia de 561.76 ms como se recoge en la Tabla 4.2. También es importante comentar que cada vez que la tasa de errores aumenta la diferencia entre ambos modelos se hace más notoria.

Tabla 4.2: Diferencias de DCT de QUIC entre PER uniforme y PER GE con N_b=20

		PER uniforme	GE	Diferencia
PER	0.01	4855.72	4805.43	50.29
	0.02	4941.15	4818.84	122.31
	0.03	5040.02	4849.26	190.76
	0.05	5478.74	4916.98	561.76

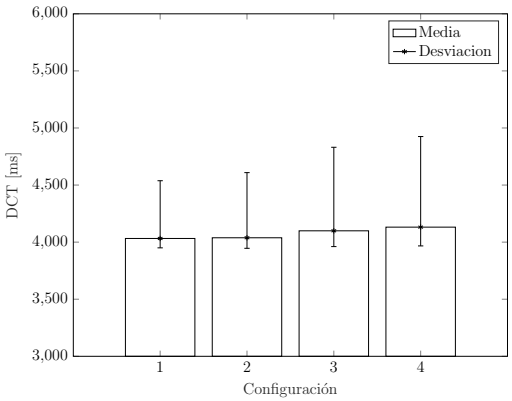
4.2.2. rQUIC

En este apartado se analizan los resultados de rQUIC con un modelo de PER uniforme y a ráfagas. Este a su vez estará diferenciado en dos subapartados: uno para rQUIC con FEC dinámico y otro para rQUIC con FEC estático.

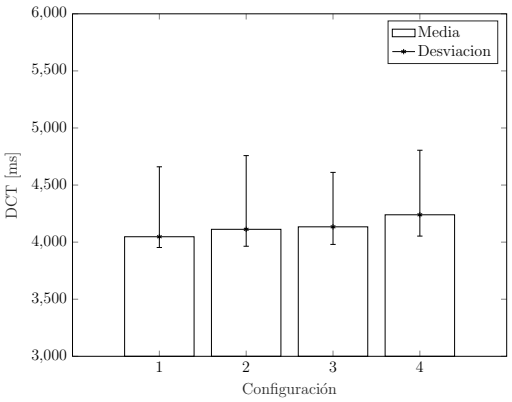
4.2.3. FEC estático

En este subapartado se exponen los resultados de rQUIC con FEC estático . En las Figuras 4.4a, 4.4b, 4.4c y 4.1d se analizan la media de las simulaciones y los valores máximos y mínimos obtenidos para los distintos valores de PER, que son los mismos que en apartado anterior.

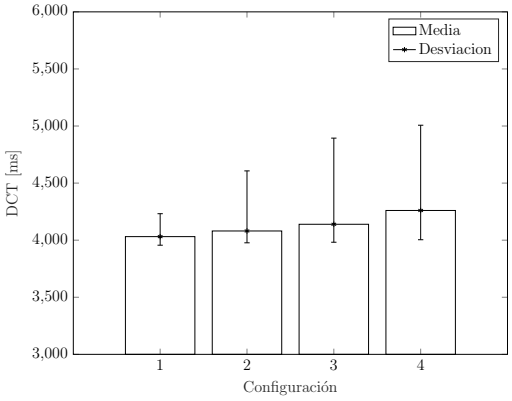
Análogamente al apartado anterior, si se comparan los valores de DCT variando el número medio de paquetes en Bad se observa que para un mayor número de paquetes, el DCT también disminuye, como se ve en la Figura 4.5, pero no de una forma tan clara como ocurría con QUIC.



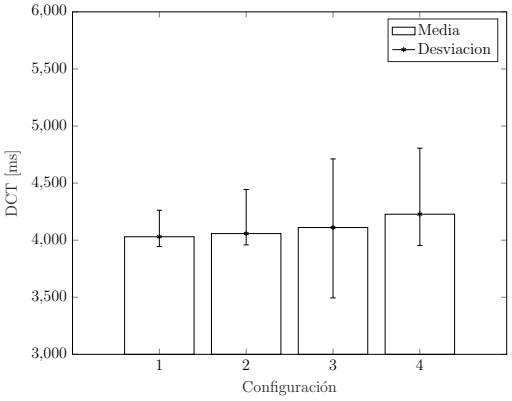
(a) rQUIC con FEC estático con PER uniforme



(b) rQUIC con FEC estático PER a ráfagas $N_b = 5$



(c) rQUIC con FEC estático con PER a ráfagas $N_b = 10$



(d) rQUIC con FEC estático con PER a ráfagas $N_b = 20$

Si se comparan los resultados obtenidos para una PER uniforme y los mejores resultados en el caso del modelo de Gilbert-Elliot se advierte una disminución del DCT en el modelo de PER uniforme como se muestra en la Figura 4.6 siendo la 'Configuración 1' para PER uniforme y la 'Configuración 2' para el modelo a ráfagas. Por ejemplo, en el caso con un mayor número de paquetes y con PER del 5 % el valor de DCT es de 4227.56 ms, mientras que para la misma tasa de error en el modelo de PER uniforme dicho valor es de 4131.41 ms, lo que hace una diferencia de 96.15 ms a favor del modelo de PER uniforme.

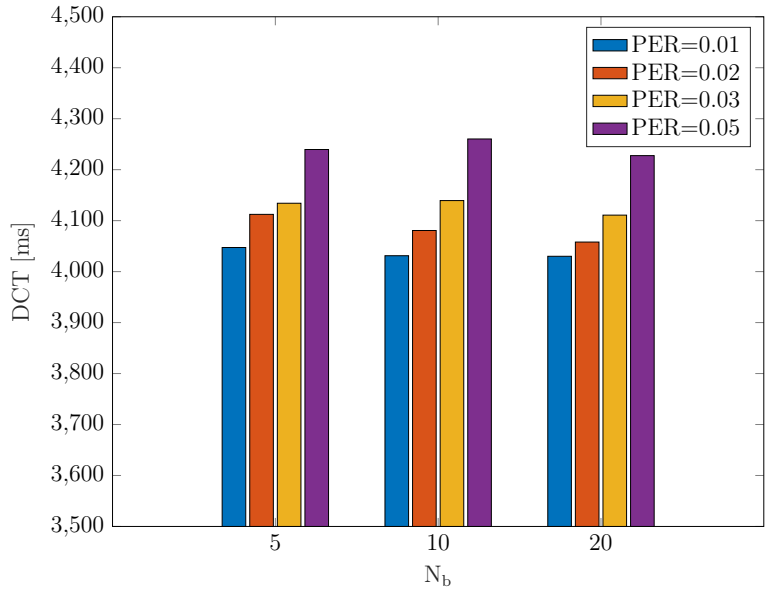


Figura 4.5: Comparación rQUIC estático modelo a ráfagas para distintos PER

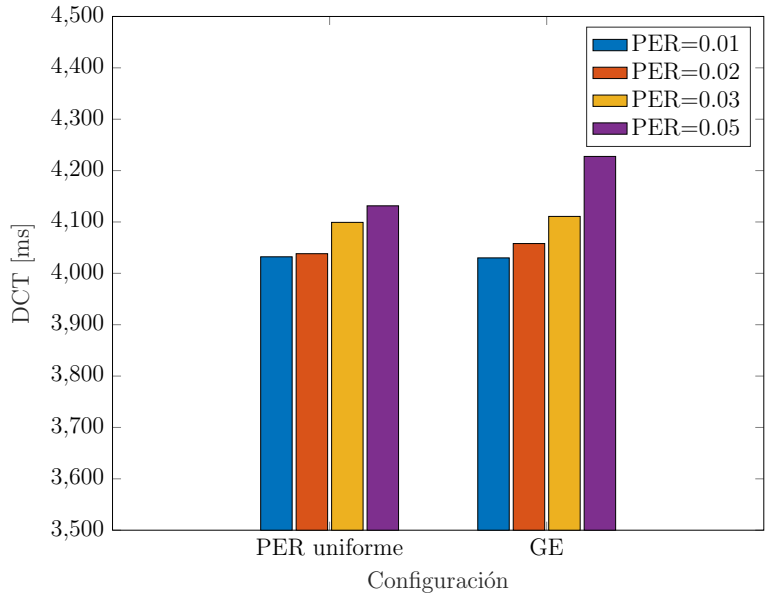


Figura 4.6: Comparación entre mejores valores de Gilbert-Elliot y PER uniforme

Tabla 4.3: Diferencias de DCT de rQUIC entre PER uniforme y PER GE con N_b=20

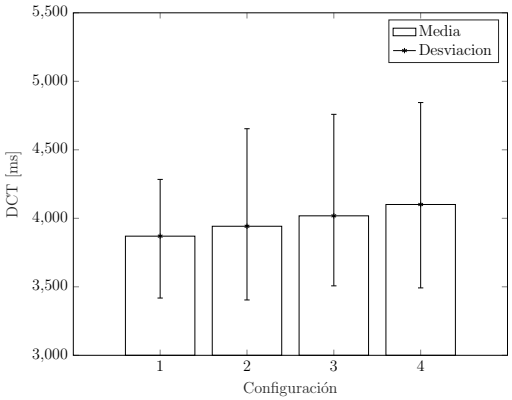
		PER uniforme	GE	Diferencia
PER	0.01	4032.18	4030.07	-2.11
	0.02	4038.23	4057.98	19.75
	0.03	4099.21	4110.81	11.6
	0.05	4131.41	4227.56	96.15

4.2.4. FEC dinámico

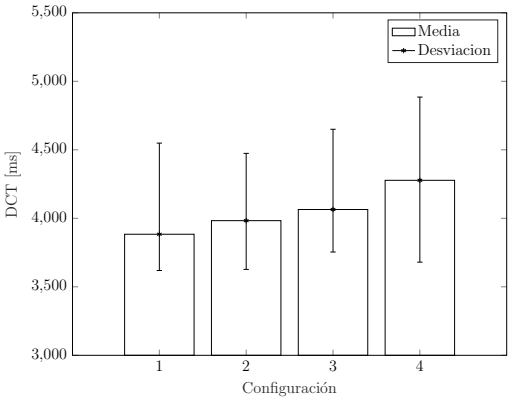
En este subapartado se analizan los resultados de rQUIC con FEC dinámico. Como en los dos apartados anteriores, se exponen los valores medios de las simulaciones y sus valores máximos y mínimos para los distintos valores de PER en las Figuras 4.7a, 4.7b, 4.7c y 4.7d.

A continuación, se compara el funcionamiento de rQUIC en el modelo a ráfagas con tres configuraciones que son para valores de N_b igual a 5, 10 y 20. Estas medidas están recogidas en la Figura 4.8, y en ella se observa como a medida que aumenta el número medio de paquetes el DCT disminuye, haciendose más evidente cuanto mayor es la tasa de error. Un ejemplo de ello sería la diferencia entre el DCT para N_b=5 y N_b=20, ambos con una PER del 5 %, donde los valores, respectivamente, son de 4277.37 ms y 4051.24 ms dando lugar a una diferencia de 226.13 ms.

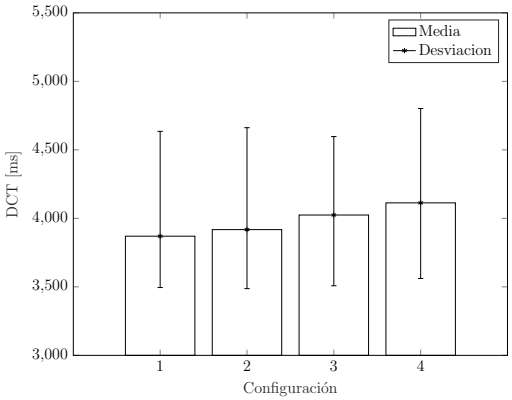
En las Figuras 4.9 y 4.10 se compraran los resultados de PER uniforme con los mejores y peores resultados, respectivamente, en el modelo de GE. En la primera de ellas, los mejores resultados de GE se corresponden con los valores de $N_b = 20$ y se observan que estos son mejores que los resultados para una PER uniforme. Sin embargo, en la segunda gráfica se observa que los peores valores de GE son peores también que los obtenidos para una PER uniforme. Estos son los correspondientes a N_b=5. Por consiguiente, estos resultados hacen parecer que rQUIC con FEC dinámico funcionan mejor en un entorno con mayor comportamiento a ráfagas.



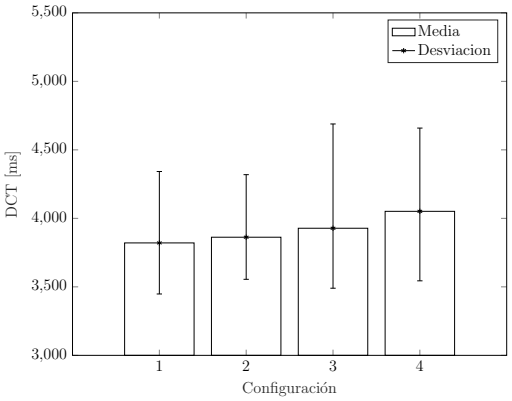
(a) rQUIC con FEC dinámico con PER uniforme



(b) rQUIC con FEC dinámico PER a ráfagas $N_b = 5$



(c) rQUIC con FEC dinámico con PER a ráfagas $N_b = 10$



(d) rQUIC con FEC dinámico con PER a ráfagas $N_b = 20$

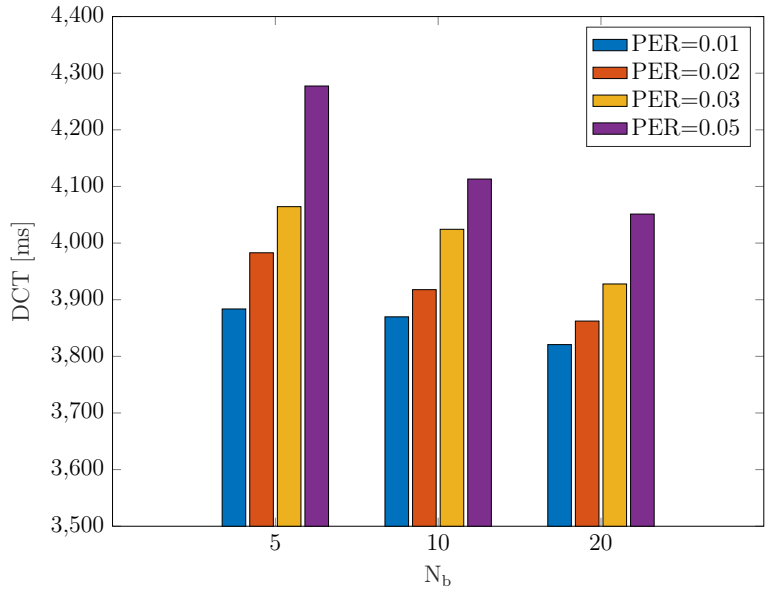


Figura 4.8: Comparación rQUIC dinámico modelo a ráfagas para distintos PER

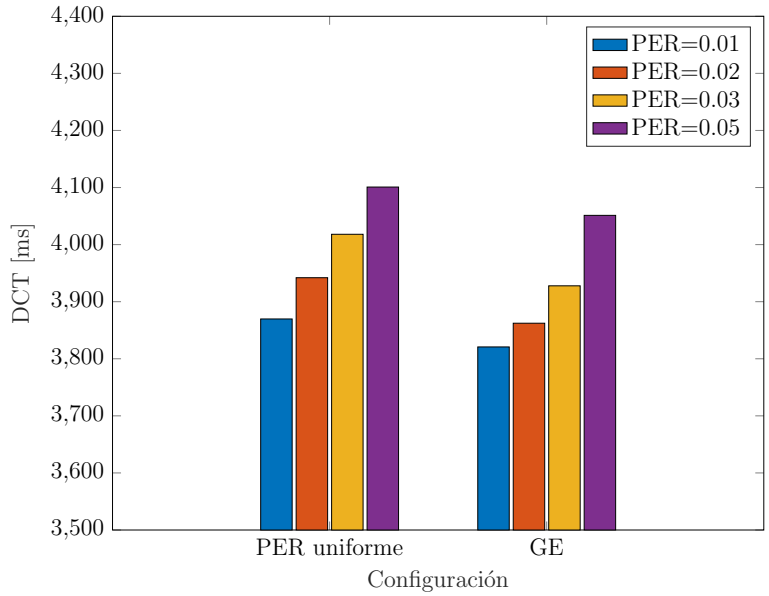


Figura 4.9: Comparación entre mejores valores de Gilbert-Elliot y PER uniforme

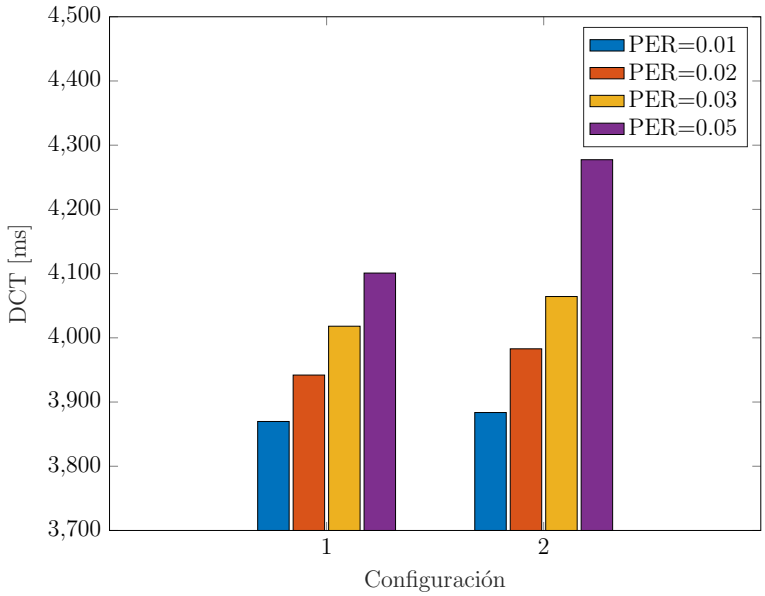


Figura 4.10: Comparación entre peores valores de Gilbert-Elliot y PER uniforme

4.2.5. Comparación general

En este apartado se van a analizar los resultados que ya se han expuesto en los apartados anteriores, pero desde otro punto de vista. En este caso se comparan QUIC, rQUIC con FEC estático y rQUIC con FEC dinámico en los dos modelos de errores.

PER uniforme

En esta sección se analizan los diferentes protocolos para un modelo de errores con PER uniforme.

Como se aprecia en la Figura 4.11, donde las configuraciones se corresponden con QUIC, rQUIC dinámico y rQUIC estático respectivamente, cualquiera de los dos modelos de rQUIC responde mejor al modelo de PER uniforme que QUIC.

Analizandolo más en detalle con los valores de la Tabla 4.4, se observa que la mayor diferencia entre QUIC y rQUIC la encontramos para una PER del

5 % y es de unos 1300 ms, concretamente de 1347.33 ms para rQUIC estático y de 1377.89 ms para rQUIC dinámico. Entre los dos tipos de rQUIC se ve que el DCT en rQUIC dinámico es ligeramente menor, con una diferencia de 162.45, 20.14, 81.12 y 30.56 ms para cada uno de los valores de PER de menor a mayor.

Tabla 4.4: DCT en ms para PER uniforme

		QUIC	rQUIC estático	rQUIC dinámico
PER	0.01	4855.72	4032.18	3869.73
	0.02	4941.75	4038.23	3942.01
	0.03	5040.02	4099.21	4018.09
	0.05	5478.74	4131.41	4100.85

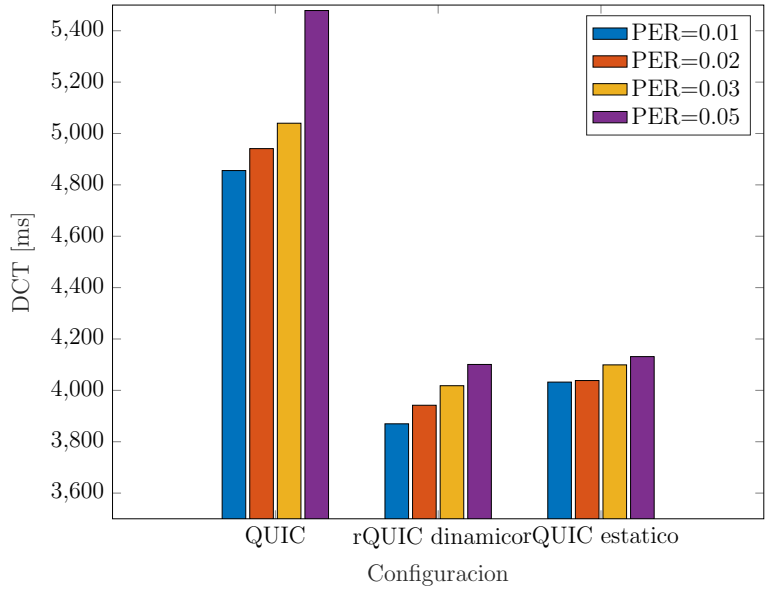


Figura 4.11: DCT en FEC uniforme

PER a ráfagas

En esta sección se analizan los resultados de los diferentes protocolos para el modelo de Gilbert-Elliott y para los valores de N_b igual a 5, 10 y 20.

Como se puede apreciar en la Figura 4.12, donde el orden de las configuraciones es el mismo que el de la sección anterior, para el valor de N_b=5 cualquiera de los dos modelos de rQUIC responde mejor al modelo a ráfagas que QUIC, al igual que pasaba con el modelo de PER uniforme.

Analizando los valores de la Tabla 4.5, se observa que la mayor diferencia se encuentra, como en el caso anterior con la PER del 5 %. Comparando los valores de los dos rQUIC, se observa que para todos los valores de PER con la excepción del 5 % los valores óptimos son los del rQUIC dinámico con una diferencia de 228.65, 151.25, 175.25 ms, desde el valor más pequeño de PER al mayor. Sin embargo, para el valor de 5 % el DCT del rQUIC estático es 46.2 ms menor que el del rQUIC dinámico.

Tabla 4.5: DCT en ms para PER a ráfagas con N_b=5

		QUIC	rQUIC estático	rQUIC dinámico
PER	0.01	4815.84	4112.31	3883.66
	0.02	4822.99	4134.13	3982.88
	0.03	4866.13	4239.57	4064.32
	0.05	4947.93	4231.17	4277.37

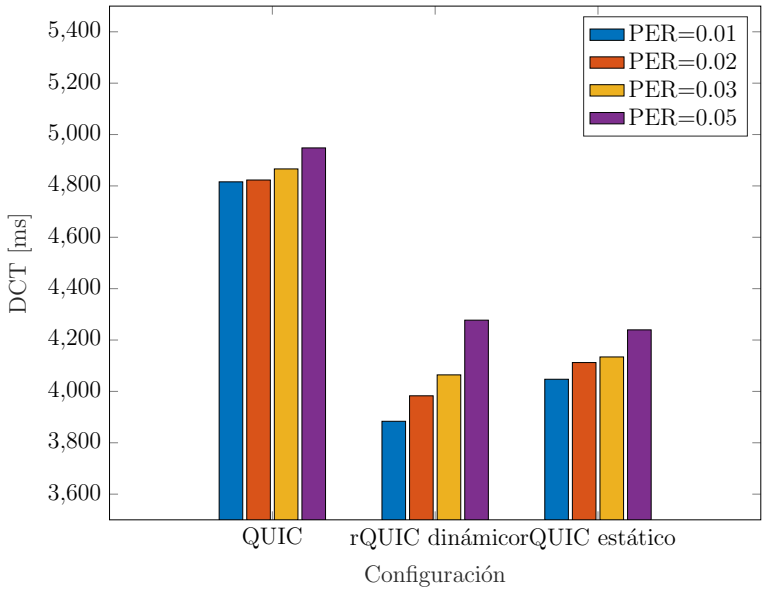


Figura 4.12: DCT en FEC a ráfagas con N_b=5

Para el valor de N_b=10, en la Figura 4.13 se observa de nuevo que los valores de DCT de QUIC para cualquier PER son mayores que los valores de DCT en rQUIC. Por otra parte, los si comparamos entre rQUIC estático y dinámico los valores para este último son menores.

En detalle, analizando los valores de la Tabla 4.6, se observa que la diferencia entre el DCT en QUIC y rQUIC es de unos 800 ms. A su vez, las diferencias entre el DCT para los distintos valores de PER, de menor a mayor, entre rQUIC dinámico y estático son de 161.41, 162.91, 115.01 y 147.13 ms.

Tabla 4.6: DCT en ms para PER a ráfagas con N_b=10

		QUIC	rQUIC estático	rQUIC dinámico
PER	0.01	4810.83	4031.17	3869.76
	0.02	4829.35	4080.66	3917.75
	0.03	4870.9	4139.36	4024.35
	0.05	4960.1	4260.22	4113.09

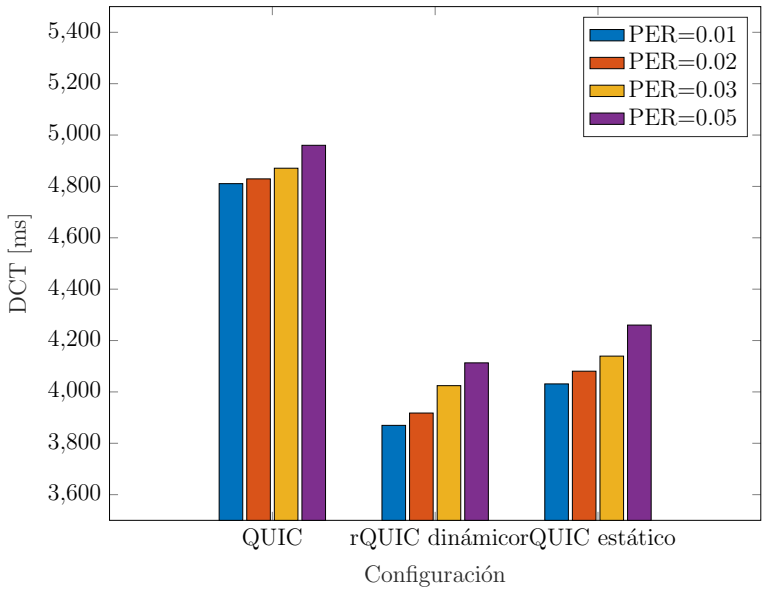


Figura 4.13: DCT en FEC a ráfagas con N_b=10

Por último, para N_b=20 si comparamos QUIC y rQUIC pasa lo mismo que en todos los casos anteriores, donde los valores de DCT para rQUIC

son mejores que para QUIC. Además, comparando entre rQUIC estático y rQUIC dinámico, este último tiene es que mejores valores de DCT tiene para cualquier valor de PER, como se muestra en la Figura 4.14.

Analizando los valores de la Tabla 4.7, se observa que la diferencia entre el DCT de QUIC y rQUIC es de entre 900 y 700 ms y la diferencia entre rQUIC estático y dinámico para cada uno de los cuatro valores de PER es de 209.34, 195.69, 182.91 y 176.32 ms respectivamente.

Tabla 4.7: DCT en ms para PER a ráfagas con N_b=20

		QUIC	rQUIC estático	rQUIC dinámico
PER	0.01	4805.43	4030.07	3820.73
	0.02	4818.84	4057.98	3862.29
	0.03	4849.26	4110.81	3927.9
	0.05	4916.98	4227.56	4051.24

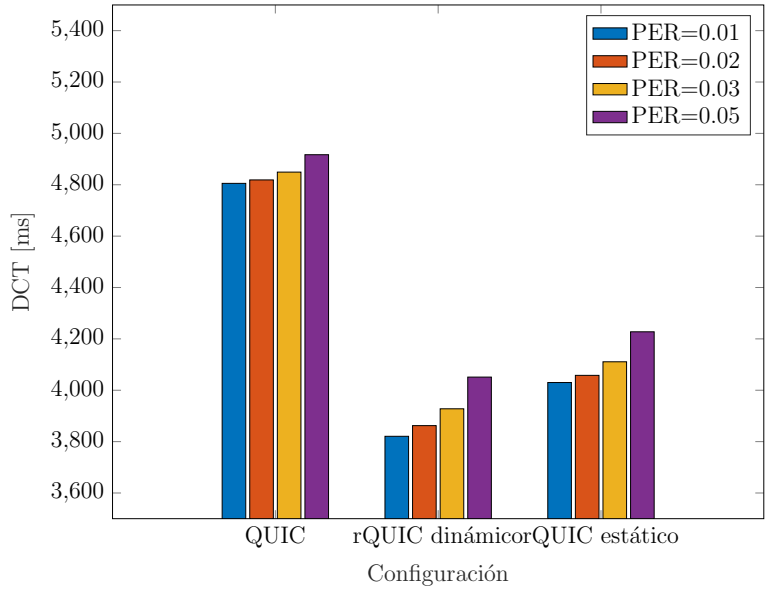


Figura 4.14: DCT en FEC a ráfagas con N_b=20

Capítulo 5

Conclusiones y líneas futuras

Con este último capítulo se finaliza la memoria, exponiendo las conclusiones a las que se ha llegado y resumiendo las comparaciones de los diferentes protocolos en los diferentes modelos. Además, se expondrán brevemente las posibles líneas futuras que surgen a partir de los resultados obtenidos en este trabajo.

5.1. Conclusiones

Con el tiempo, el ancho de banda en todo el mundo creecerá, pero los tiempos de ida y vuelta, limitados en última instancia por la velocidad de la luz, no disminuirán. Por ello se necesita un protocolo para enviar solicitudes, respuestas y cualquier tipo de interacción a través de internet con menos latencia y con menos retransmisiones y, siendo este el principal objetivo de QUIC.

En este trabajo, se ha analizado el comportamiento de QUIC y rQUIC (añade FEC) sobre dos escenarios.

En primer lugar, se observa que QUIC y rQUIC con FEC dinámico funcionan mejor en un entorno a ráfagas que en un entorno con PER uniforme. En el caso de QUIC, el DCT se reduce hasta en un 10.25 % para el modelo a ráfagas y, en el caso de rQUIC dinámico, el DCT se reduce en un 1.2 %. Sin

embargo, para el caso de rQUIC dinámico, esto solo ocurre para casos en los que número medio de paquetes es elevado. De esta manera, se aprecia que cuando el número medio de paquetes no es muy alto rQUIC funciona mejor con una PER uniforme que un entorno a ráfagas. En estos casos, cogiendo el valor de N_b=5 y con una PER del 5 %, el DCT en un modelo con PER uniforme es un 4.13 % menor que en un entorno a ráfagas.

En el caso de rQUIC con FEC estático, se observa que responde mejor sobre un canal PER uniforme que un entorno a ráfagas, independientemente del número medio de paquetes y de la PER. En este caso los resultados para un entorno de PER uniforme son un 2.27 % mejores que esobre un canal a ráfagas.

En segundo lugar, comparando los distintos protocolos se observa que rQUIC responde mejor que QUIC en cualquiera de los dos entornos. En el caso del escenario con PER uniforme el DCT se reduce hasta un 25.15 % con rQUIC dinámico frente a QUIC. Por otra parte, rQUIC con FEC dinámico y estático tienen DCT muy similares. Para el caso del modelo a ráfagas, los valores para rQUIC son un 14.5 %, 17.1 % y 17.6 % menor que QUIC para los valores de N_b igual a 5, 10 y 20, respectivamente. En el caso de N_b=5 es el único en el que rQUIC con FEC estático es mejor que rQUIC con FEC dinámico, pero únicamente para una PER del 5 %. En el resto de casos el DCT en rQUIC dinámico es menor que en el estático, un 3.5 % y un 4.17 %, respectivamente.

Según lo expuesto anteriormente, rQUIC dinámica seria la mejor solución para entornos a ráfagas con una alta congestión y tasas de pérdida de paquetes altas.

5.2. Líneas futuras

En esta sección se exponen algunas de las posibles líneas futuras que se aparecen a partir de los resultados aportados por este trabajo.

Por una parte, se podría seguir investigando en entornos a ráfagas y profundizando en los resultados y, caracterizando de manera definitiva las direncias en esos mismos resultados.

Otra posible línea para investigar sería el análisis en profundidad del motivo de que los resultados sean mejores en sistemas con una mayor congestión, introduciendo en el estudio el control de flujo y de congestión.

Anexos A

Código C++ Modelo de errores de Gilbert-Elliot

```
#ifndef GILBERTELLIOT_H
#define GILBERTELLIOT_H


#include "ns3/object.h"
#include "ns3/random-variable-stream.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/pointer.h"
#include "ns3/double.h"
#include <time.h>
#include <math.h>

namespace ns3 {

    class Gilbert_Elliot : public ErrorModel {
    public:
        static TypeId GetTypeId(void);

        Gilbert_Elliot();

        ~Gilbert_Elliot();


        TracedValue<double> m_per;
        TracedValue<double> m_tiempo;
        TracedValue<double> m_dct;
        double m_primero;
```

```
        double m_previo;
        double adicional;
    private:

        bool DoCorrupt(ns3::Ptr<ns3::Packet> p);

        bool anterior;
        double correctos;
        double incorrectos;
        double r; //Probabilidad de ir Good --> Bad
        double q; //Probabilidad de ir Bad --> Good
        int i;

        double stats(double correctos, double incorrectos);

        void DoReset(void);

    };

    NS_OBJECT_ENSURE_REGISTERED(Gilbert_Elliot);
    //     NS_LOG_COMPONENT_DEFINE("GilbertElliot");

    TypeId Gilbert_Elliot::GetTypeId(void) {
        static TypeId tid = TypeId("ns3::GilbertElliot")
                .SetParent<ErrorModel> ()
                .AddAttribute("PrGood", "Probabilidad de ir Good
                    --> Bad",
                DoubleValue(),
                MakeDoubleAccessor(&Gilbert_Elliot::r),
                MakeDoubleChecker<double> ())
                .AddAttribute("PrBad", "Probabilidad de ir Bad
                    --> Good",
                DoubleValue(),
                MakeDoubleAccessor(&Gilbert_Elliot::q),
                MakeDoubleChecker<double> ())
                .AddTraceSource("PER", "Packet Error Rate",
                MakeTraceSourceAccessor(&Gilbert_Elliot::m_per),
                "ns3::TracedValueCallback::Double")
                .AddTraceSource("Tiempo", "Tiempo",
                MakeTraceSourceAccessor(&Gilbert_Elliot::
                    m_tiempo),
                "ns3::TracedValueCallback::Double")
                .AddTraceSource("DCT", "Download Complete Time",
                MakeTraceSourceAccessor(&Gilbert_Elliot::m_dct),
                "ns3::TracedValueCallback::Double")
                ;
        return tid;
    }
```

```
Gilbert_Elliot::Gilbert_Elliot()
: m_per(0)
, m_tiempo(0)
, anterior(true)
, correctos(0)
, incorrectos(0)
, i(0)
, m_previo(0)
, adicional(0)
, m_dct(0) {

}

Gilbert_Elliot::~Gilbert_Elliot() {
}

double Gilbert_Elliot::stats(double correctos, double
    incorrectos) {
    std::cout << "Correctos: " << correctos << "; "
        Incorrectos: " << incorrectos << std::endl;
    auto per = incorrectos / (correctos + incorrectos);
    return per;
}

bool
Gilbert_Elliot::DoCorrupt(Ptr<Packet> p) {
    std::cout << "Recibido paquete " << p->GetUid() << std::
        endl;

    Ptr<UniformRandomVariable> uniform1 = CreateObject<
        UniformRandomVariable> ();
    Ptr<UniformRandomVariable> uniform2 = CreateObject<
        UniformRandomVariable> ();

    double value1, value2;
    const double min = 0;
    const double max = 1;
    value1 = uniform1->GetValue(min, max);
    value2 = uniform2->GetValue(min, max);


    if (p->GetSerializedSize() == 128 || p->
        GetSerializedSize() == 380) //Para RQUIC = 1308; Para
         QUIC != 128 and 380
    {
        if (anterior == false)
        {
```

```
                if (value2 > (1 - q))
                {
                    anterior = true;
                    correctos = correctos + 1;
                    std::cout << "BAD -> GOOD" << std::endl;
                    m_per = stats(correctos, incorrectos);
                    m_tiempo = Simulator::Now().GetMilliSeconds
                        () - adicional;
                    std::cout << "-------------------------" <<
                       std::endl;

                    if (i == 0)
                    {
                        m_primero = m_tiempo;
                    }

                }
                else
                {
                    incorrectos = incorrectos + 1;
                    std::cout << "BAD -> BAD" << std::endl;
                    m_per = stats(correctos, incorrectos);
                    m_tiempo = Simulator::Now().GetMilliSeconds
                        () - adicional;
                    std::cout << "-------------------------" <<
                       std::endl;
                    if (i == 0)
                    {
                        m_primero = m_tiempo;
                    }
                }

            } else if (anterior == true)
            {
                if (value1 > r)
                {
                    correctos = correctos + 1;
                    std::cout << "GOOD -> GOOD" << std::endl;
                    m_per = stats(correctos, incorrectos);
                    m_tiempo = Simulator::Now().GetMilliSeconds
                        () - adicional;
                    std::cout << "-------------------------" <<
                       std::endl;
                    if (i == 0)
                    {
                        m_primero = m_tiempo;
                    }
                } else
```

```
                {
                    anterior = false;
                    incorrectos = incorrectos + 1;
                    std::cout << "GOOD -> BAD" << std::endl;
                    m_per = stats(correctos, incorrectos);
                    m_tiempo = Simulator::Now().GetMilliSeconds
                        () - adicional;
                    std::cout << "--------------------------" <<
                       std::endl;
                    if (i == 0)
                    {
                        m_primero = m_tiempo;
                    }
                }
            }
        std::cout << "PER = " << m_per << std::endl;
        std::cout << "Primero: " << m_primero << std::endl;
        std::cout << "Paquetes correctamente recibidos: " <<
           correctos << std::endl;
        m_tiempo = m_tiempo - m_primero;
        std::cout << "DCT " << m_tiempo << " ms" << std::endl;
        i = i + 1;
        std::cout << "Pkts: " << i  << std::endl;

        if ((m_tiempo - m_previo) > 10000 && i != 0)
        {
            m_dct = m_previo;
            std::cout << "D C T : " << m_dct << std::endl;
            adicional = adicional + m_tiempo;
            DoReset();
        }
        m_previo = m_tiempo;
        }
        return !anterior;
    }
    void
    Gilbert_Elliot::DoReset(void) {
    m_per = 0;
    m_tiempo = 0;
    anterior = true;
    correctos = 0;
    incorrectos = 0;
    i = 0;
    m_previo = 0;
    }
}
#endif /* GILBERTELLIOT_H */
```

Anexos B

Código C++ del programa principal para GE

```
#include <iostream>
#include <fstream>

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/csma-module.h"
#include "ns3/tap-bridge-module.h"

#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/stats-module.h"

#include "ns3/propagation-loss-model.h"
#include "ns3/jakes-propagation-loss-model.h"
#include "ns3/constant-position-mobility-model.h"

#include "ns3/config.h"
#include "ns3/command-line.h"
#include "ns3/string.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/pointer.h"
#include "ns3/gnuplot.h"
#include "ns3/simulator.h"
#include "GilbertElliot.h"

#include "ns3/object.h"
#include "ns3/uinteger.h"
```

```
#include "ns3/traced-value.h"
#include "ns3/trace-source-accessor.h"
#include <string>
#include "ns3/netanim-module.h"



using namespace ns3;

NS_LOG_COMPONENT_DEFINE("TapCsmaVirtualMachineExample");

void
PerTrace(Ptr<OutputStreamWrapper> stream, double oldValue,
    double newValue) {
     *stream->GetStream () << Simulator::Now().GetMilliSeconds()
        << " ms : : Traced " << oldValue << " to " << newValue <<
        std::endl;
}

static void
PerGraphics (Ptr<OutputStreamWrapper> stream2, double oldValue,
    double newValue)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" <<
      newValue);
   double tiempo = Simulator::Now ().GetSeconds ();
   *stream2->GetStream () << tiempo << "\t" << oldValue << "\t"
      << newValue << std::endl;
}
void
TimeTrace(Ptr<OutputStreamWrapper> stream3, double oldValue,
    double newValue) {
     *stream3->GetStream () << Simulator::Now().GetMilliSeconds()
         << " ms : : Traced " << oldValue << " to " << newValue <<
         std::endl;

     double tiempo, primero;
     if (oldValue == 0)
     {
         primero = newValue;
         tiempo = 0;

     }
     else
     {
         tiempo = newValue - primero;
         *stream3->GetStream() << primero << std::endl;
     }
```

```
    *stream3->GetStream () << Simulator::Now().GetMilliSeconds()
        << "ms :: DCT: " << tiempo << std::endl;
}

void
DCTTrace(Ptr<OutputStreamWrapper> stream4, double oldValue,
    double newValue) {
    *stream4->GetStream () << "DCT " << newValue << " ms" << std
        ::endl;
}


int
main(int argc, char *argv[]) {


    std::string errorModelType = "ns3::RateErrorModel";

    CommandLine cmd;
    cmd.AddValue("errorModelType", "TypeId of the error model to
        use", errorModelType);
    cmd.Parse(argc, argv);

    GlobalValue::Bind("SimulatorImplementationType", StringValue
        ("ns3::RealtimeSimulatorImpl"));
    GlobalValue::Bind("ChecksumEnabled", BooleanValue(true));

    NodeContainer nodes;
    nodes.Create(2);
    NS_LOG_INFO("Nodes created"); //New

    CsmaHelper csma;
    csma.SetChannelAttribute("Delay", StringValue("2ms"));
    NS_LOG_INFO("Delay: 2ms");




    NetDeviceContainer devices = csma.Install(nodes);
    Config::SetDefault("ns3::GilbertElliot::PrGood", DoubleValue
        (0.4));
    Config::SetDefault("ns3::GilbertElliot::PrBad", DoubleValue
        (0.75));
    auto pem = CreateObject<Gilbert_Elliot> ();

    devices.Get(1)->SetAttribute("ReceiveErrorModel",
        PointerValue(pem));
```

```
    AsciiTraceHelper asciiTraceHelper;

    Ptr<OutputStreamWrapper> stream = asciiTraceHelper.
        CreateFileStream ("GE.txt");
    pem->TraceConnectWithoutContext ("PER", MakeBoundCallback (&
        PerTrace, stream));

    Ptr<OutputStreamWrapper> stream2 = asciiTraceHelper.
        CreateFileStream ("GEGraph.txt");
    pem->TraceConnectWithoutContext ("PER", MakeBoundCallback (&
        PerGraphics, stream2));

    Ptr<OutputStreamWrapper> stream3 = asciiTraceHelper.
        CreateFileStream ("GETime.txt");
    pem->TraceConnectWithoutContext ("Tiempo", MakeBoundCallback
         (&TimeTrace, stream3));

    Ptr<OutputStreamWrapper> stream4 = asciiTraceHelper.
        CreateFileStream ("GEdct.txt");
    pem->TraceConnectWithoutContext ("DCT", MakeBoundCallback (&
        DCTTrace, stream4));


    TapBridgeHelper tapBridge;
    tapBridge.SetAttribute("Mode", StringValue("UseBridge"));
    tapBridge.SetAttribute("DeviceName", StringValue("tap-left")
        );
    tapBridge.Install(nodes.Get(0), devices.Get(0));

    tapBridge.SetAttribute("DeviceName", StringValue("tap-right"
        ));
    tapBridge.Install(nodes.Get(1), devices.Get(1));

    Simulator::Run();
    Simulator::Destroy();
}
```

Anexos C

Código C++ errores con PER uniforme

```
#ifndef PERUNIFORM_H
#define PERUNIFORM_H

#include "ns3/object.h"
#include "ns3/random-variable-stream.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/pointer.h"
#include "ns3/double.h"
#include <time.h>
#include <math.h>

namespace ns3 {

    class PER_Uniform : public ErrorModel {
    public:
        static TypeId GetTypeId(void);

        PER_Uniform();

        ~PER_Uniform();

        TracedValue<double> m_per;
        TracedValue<double> m_tiempo;
        TracedValue<double> m_dct;
        double m_primero;
        double m_previo;
        double adicional;
```

```
private:

    bool DoCorrupt(ns3::Ptr<ns3::Packet> p);
    bool anterior;
    double correctos;
    double incorrectos;
    int i;

    void DoReset(void);

};


NS_OBJECT_ENSURE_REGISTERED(PER_Uniform);

TypeId PER_Uniform::GetTypeId(void) {
    static TypeId tid = TypeId("ns3::PER_Uniform")
            .SetParent<ErrorModel> ()
            .AddAttribute("PER", "Packet Error Rate",
            DoubleValue(),
            MakeDoubleAccessor(&PER_Uniform::m_per),
            MakeDoubleChecker<double> ())
            .AddTraceSource("Tiempo", "Tiempo",
            MakeTraceSourceAccessor(&PER_Uniform::m_tiempo),
            "ns3::TracedValueCallback::Double")
            .AddTraceSource("DCT", "Download Complete Time",
            MakeTraceSourceAccessor(&PER_Uniform::m_dct),
            "ns3::TracedValueCallback::Double")
            ;
    return tid;
}

PER_Uniform::PER_Uniform()
: m_per(0)
, m_tiempo(0)
, anterior(true)
, i(0)
, m_previo(0)
, adicional(0)
, m_dct(0) {
}

PER_Uniform::~PER_Uniform() {
}

bool
PER_Uniform::DoCorrupt(Ptr<Packet> p) {
```

```
Ptr<UniformRandomVariable> uniform1 = CreateObject<
    UniformRandomVariable> ();

double value1;
const double min = 0;
const double max = 1;
value1 = uniform1->GetValue(min, max);


if (p->GetSerializedSize() == 128 || p->
    GetSerializedSize() == 380) //Para RQUIC = 1308; Para
     QUIC != 128 and 380
{
     if ( value1 > m_per )
     {
          std::cout << "Paquete recibido correctamente" <<
               std::endl;
          m_tiempo = Simulator::Now ().GetMilliSeconds ()
             - adicional;
          if (i == 0)
               {
                    m_primero = m_tiempo;
               }
     }
     else
     {
          std::cout << "Paquete perdido" << std::endl;
          m_tiempo = Simulator::Now ().GetMilliSeconds ()
             - adicional;
          if (i == 0)
               {
                    m_primero = m_tiempo;
               }
     }

m_tiempo = m_tiempo - m_primero;
std::cout << "DCT " << m_tiempo << " ms" << std::endl;
i = i + 1;
std::cout << "Pkts: " << i  << std::endl;

if ((m_tiempo - m_previo) > 10000 && i != 0)
{
     m_dct = m_previo;
     std::cout << "D C T : " << m_dct << std::endl;
     adicional = adicional + m_tiempo;
     DoReset();
}
```

```
        m_previo = m_tiempo;

        }

        return corrupt;
    }

    void
    PER_Uniform::DoReset(void) {
    m_per = 0;
    m_tiempo = 0;
    anterior = true;
    i = 0;
    m_previo = 0;
    anterior = true;
    }

}
#endif /* UNIFORM_H */
```

Anexos D

Código C++ del programa principal para PER uniforme

```
#include <iostream>
#include <fstream>

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/csma-module.h"
#include "ns3/tap-bridge-module.h"

#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/stats-module.h"

#include "ns3/propagation-loss-model.h"
#include "ns3/jakes-propagation-loss-model.h"
#include "ns3/constant-position-mobility-model.h"

#include "ns3/config.h"
#include "ns3/command-line.h"
#include "ns3/string.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/pointer.h"
#include "ns3/gnuplot.h"
#include "ns3/simulator.h"
#include "Uniform.h"

#include "ns3/object.h"
#include "ns3/uinteger.h"
```

```
#include "ns3/traced-value.h"
#include "ns3/trace-source-accessor.h"

#include <string>

#include "ns3/netanim-module.h"


using namespace ns3;

NS_LOG_COMPONENT_DEFINE("TapCsmaVirtualMachineExample");


void
DCTTrace(Ptr<OutputStreamWrapper> stream4, double oldValue,
    double newValue) {
    //*stream4->GetStream () << "DCT: " << newValue << " ms";
    *stream4->GetStream () << "DCT " << newValue << " ms" << std
        ::endl;
}


int
main(int argc, char *argv[]) {

    std::string errorModelType = "ns3::RateErrorModel";

    CommandLine cmd;
    cmd.AddValue("errorModelType", "TypeId of the error model to
        use", errorModelType);
    cmd.Parse(argc, argv);

    GlobalValue::Bind("SimulatorImplementationType", StringValue
        ("ns3::RealtimeSimulatorImpl"));
    GlobalValue::Bind("ChecksumEnabled", BooleanValue(true));

    NodeContainer nodes;
    nodes.Create(2);
    NS_LOG_INFO("Nodes created"); //New

    CsmaHelper csma;
    csma.SetChannelAttribute("Delay", StringValue("2ms"));
    NS_LOG_INFO("Delay: 2ms");


    NetDeviceContainer devices = csma.Install(nodes);
    Config::SetDefault("ns3::PER_Uniform::PER", DoubleValue
        (0.02));
```

```
    auto pem = CreateObject<PER_Uniform> ();
    devices.Get(1)->SetAttribute("ReceiveErrorModel",
        PointerValue(pem));

    AsciiTraceHelper asciiTraceHelper;

    Ptr<OutputStreamWrapper> stream4 = asciiTraceHelper.
        CreateFileStream ("FU_dct.txt");
    pem->TraceConnectWithoutContext ("DCT", MakeBoundCallback (&
        DCTTrace, stream4));

    TapBridgeHelper tapBridge;
    tapBridge.SetAttribute("Mode", StringValue("UseBridge"));
    tapBridge.SetAttribute("DeviceName", StringValue("tap-left")
        );
    tapBridge.Install(nodes.Get(0), devices.Get(0));

    tapBridge.SetAttribute("DeviceName", StringValue("tap-right"
        ));
    tapBridge.Install(nodes.Get(1), devices.Get(1));

    Simulator::Run();
    Simulator::Destroy();
}
```

Bibliografía

[1] O. Nalawade, A. Dhanwani, and T. Prabhu, "Comparison of present-day transport layer network protocols and google's quic," in *2018 International Conference on Smart City and Emerging Technology (ICSCET)*, pp. 1–8, Jan 2018.

[2] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz, "Multipath quic: A deployable multipath transport protocol," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–7, May 2018.

[3] F. Michel, Q. De Coninck, and O. Bonaventure, "Quic-fec: Bringing the benefits of forward erasure correction to quic," in *2019 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, May 2019.

[4] K. Nepomuceno, I. N. d. Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, and D. Sadok, "Quic and tcp: A performance evaluation," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00045–00051, June 2018.

[5] S. F. R. A. z. A. Pablo Garrido, Isabel Sánchez, "rquic - integrating fec with quic for robust wireless communications," 2019.

[6] J. H. Kang, W. E. Stark, and A. O. Hero, "Turbo codes for fading and burst channels," in *IEEE Conference Theory Mini Conference*, pp. 40–45, 1998.

[7] G. Hasslinger and O. Hohlfeld, "The gilbert-elliott model for packet loss in real time services on the internet," in *14th GI/ITG Conference - Measurement, Modelling and Evalutation of Computer and Communication Systems*, pp. 1–15, March 2008.

[8] A. Bildea, O. Alphand, F. Rousseau, and A. Duda, "Link quality estimation with the gilbert-elliot model for wireless sensor networks," in *2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 2049–2054, Aug 2015.